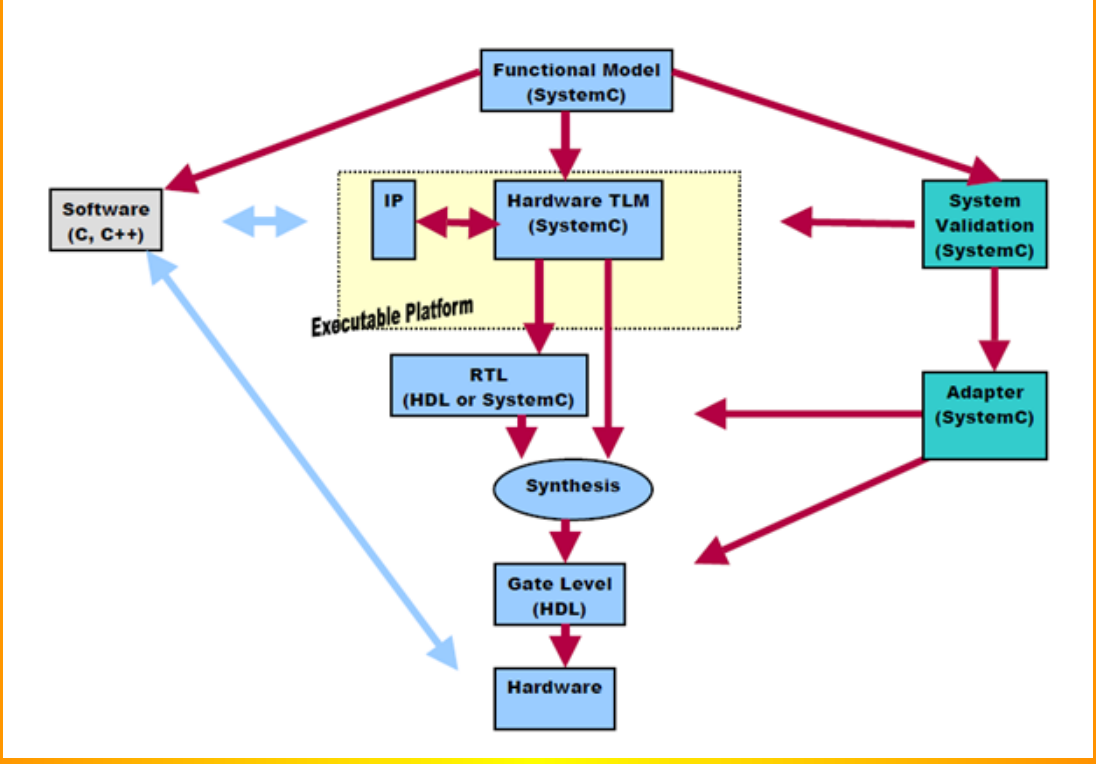
В.А. АЛЕХИН

SystemC

Моделирование электронных систем



МОСКВА 2017

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ
(МИРЭА)

В.А. Алехин

SystemC

Моделирование электронных систем

МОСКВА 2017

УДК 004.414.2.

А49

Автор: В.А. Алехин.

Учебное пособие посвящено изучению языка SystemC, который находит все более широкое применение для моделирования сложных электронных систем на разных уровнях абстрации: системное описание, уровнь транзакций, уровень регистровых пересылок и т.д. Являясь надстройкой к языку C++, SystemC успешно применяется для проектирования «Систем на кристалле», верификации сложных систем, разработок смешанных аналогово-цифровых систем.

Для практического освоения работы в SystemC пособие содержит методику установки библиотек, подробное описание языка, многочисленные примеры программ с решениями в средах Eclipse и Microsoft Visual Studio. Отдельная глава посвящена описанию моделирования на уровне транзакций TLM-2.0.

Приложение содержит справочные сведения о языке C++ и методику установки SystemC в среде Microsoft Visual Studio.

Учебное пособие предназначено для бакалавров и магистров, обучающихся по направлению «Информатика и вычислительная техника» и изучающих дисциплину «Технологии проектирования устройств и систем вычислительной техники средствами САПР».

Учебное пособие может быть полезно разработчикам современных электронных систем.

Рецензенты: доктор технических наук, профессор М.Л. Белов,

доктор технических наук, профессор Л.А. Потапов

Оглавление

Глава 1. Проблемы проектирования систем на кристалле..... 13
1.1. Введение..... 13
1.2. Цели SystemC..... 16
1.3. Уровни моделирования в SystemC..... 20
1.4. Краткая история создания и развития SystemC..... 22
1.5. Методология проектирования SystemC..... 24
1.6. Стандартные графические обозначения..... 28
1.7. Как изучать эту книгу..... 29
Глава 2. Установка SystemC в интерактивных средах разработки..... 31
Eclipse и Micrisoft Visual Studio..... 31
2.1. Установка SystemC в Eclipse с компилятором Cygwin..... 32
2.1.1. Краткие сведения об Eclipse..... 32
2.1.2. Cygwin..... 33
2.1.3. Установка Cygwin..... 33
2.1.4. Загрузка программы SystemC-2.3.1..... 38
2.1.5 Компиляция SystemC-2.3.1 в Cigwin64..... 38
2.1.6. Создание переменных сред для SystemC..... 40
2.1.7. Запуск Eclipse и настройки рабочего пространства..... 40
2.1.8. Создание нового проекта C++..... 44
2.1.9. Создание проекта «Hello-SystemC»..... 51
2.2. Как работать в Eclipse CDT..... 53
2.2.1. Главное меню, перспективы, workspace..... 53
2.2.2. Перспективы в Eclipce..... 55
2.3. Программа GTKWave..... 57
2.3.1. Главное окно GTKWave..... 59
2.3.2. Маркеры и масштабы..... 61
2.3.3. Главное меню..... 62
2.3.4. Установка и использование GTKWave..... 64
2.4. Установка SystemC-2.3.1 в операционной системе..... 67

Ubuntu 16.04 67
2.4.1. Полезные команды для работы в терминале Ubuntu 68
2.4.2. Установка SystemC-2.3.1a 70
2.4.3. Установка JAVA в Ubuntu 72
2.4.4. Установка Eclipse в Ubuntu 72
2.4.5. Установка GTKWave 74
2.5. Об установке SystemC в Microsoft Visual Studio 75
Глава 3. Основы языка SystemC-2.3.1 76
3.1. SystemC – надстройка к языку C++ 76
3.2. Ядро моделирования (Kernel) 77
3.3. Состав ядра языка SystemC (Core Language) 78
3.4. Инициализация процесса 81
3.5. Модель времени в SystemC 82
3.6. Модули SC_MODULE 83
3.6.1. Порты модулей 85
3.6.2. Сигналы модуля 86
3.6.3. Создание экземпляров модулей 86
3.6.4. Внутренние переменные 88
3.7. Конструктор SC_CTOR 89
3.8. Альтернативные конструкторы: SC_HAS_PROCESS 90
3.9. Процессы 91
3.9.1. Процесс SC_THREAD 97
3.9.2. Процесс SC_METHOD 102
3.9.3. Процесс SC_CTHREAD 106
3.10. Глобальное и локальное наблюдение 113
3.11. Порты и сигналы 126
3.11.1. Чтение и запись портов и сигналов 128
3.11.2. Массив порты и сигналы 131
3.11.3. Привязка сигналов 133
3.11.4. Разрешенные логические векторы 139

3.11.5. Разрешенные векторные сигналы..... 140
3.12. Тактирование 143
3.13. Время 148
3.14. События..... 156
3.14.1. Функция wait ()..... 160
3.14.2. Next_trigger ()..... 163
3.15. Методы 166
3.15.1. Методы sc_start() и sc_stop() 166
3.15.2. Метод wait() 166
3.15.3. Метод sc_time_stamp()..... 167
3.16. Динамическая чувствительность для SC_METHOD: next_trigger () 167
3.17. Статическая чувствительность для процессов..... 168
3.18. Типы данных и операторы 170
3.18.1. Тип sc_bit..... 171
3.18.2. Тип sc_logic..... 173
3.18.3. Битовый вектор произвольной длины sc_bv 176
3.18.4. Беззнаковые и знаковые целые..... 178
с фиксированной точностью sc_int <n> и sc_uint <n>..... 178
3.18.5. Знаковые и беззнаковые типы целых с произвольной..... 183
точностью sc_bigint и sc_biguint..... 183
3.18.6. Логический вектор произвольной длины sc_lv <N> 185
3.19. Оператор сравнения..... 186
3.20. Трассировка определенного пользователем типа..... 186
3.22. Типы с фиксированной точкой..... 187
3.23. Каналы и интерфейсы..... 189
3.23.1. Канал sc_mutex 189
3.23.2. Канал sc_semaphore..... 193
3.23.3. Канал sc_fifo 197
3.23.4. Иерархические каналы 200
3.24. Коммуникации в SystemC 201

3.25. Моделирование уровня транзакций 203
3.26. Моделирование и отладка с помощью SystemC 203
3.26.1. Планировщик SystemC 204
3.26.2. Контроль моделирования 205
3.26.3. Расширенные методы техники контроля моделирования 205
3.27. Трассировка осциллограмм 207
3.27.1. Создание файла трассировки 207
3.27.2. Трассировка скалярной переменной и сигналов 208
3.27.3. Трассировка переменных и сигналов совокупного типа 209
3.27.4. Трассировка переменных и массивов сигналов 210
3.27.5. Отладка SystemC 210
Глава 4. Практическое программирование в SystemC 211
4.1. Введение 211
4.2. Два основных стиля 211
4.3. Традиционный шаблон 211
4.4. Рекомендуемая альтернативная форма шаблона 213
4.5. Описание библиотек SystemC 214
4.6. Еще одно приветствие в SystemC 215
4.7. Базовый пример канала связи для сложных моделей 217
4.8. Использование SystemC для RTL синтеза устройств 229
4.9. Испытательные программы Testbench 234
4.9.1. Основные конструкции испытательных программ 234
4.9.2. Сигналы 238
4.10. Пример моделирования D-триггера с испытательной 243
программой 243
4.11. Программы «First_counter» и «Testbench» 247
4.12. Пример мультиплексора MUX 253
4.13. Базовый пример моделирования простой шины на уровне транзакций 256
Глава 5. Моделирование на уровне транзакций в SystemC 263
5.1 Введение 263

5.2. Концепции моделирования 263
5.3. Инициаторы, цели и сокеты 264
5.4. Общая полезная нагрузка и блокирующий транспорт 267
5.5. Временная аннотация 271
5.6. Взаимодействие и базовый протокол 272
5.7. Интерфейсы TLM-2.0 273
5.8. Исходный код и документация 275
5.9. Моделирование на уровне транзакций, варианты 275
использования и абстракция 275
5.10. Стили кодирования 277
5.10.1. Стиль untimed 277
5.10.2. Стиль Loosely-timed и временная развязка 278
5.10.3. Характеристика свободно-временных и приближенно-временных стилей кодирования 280
5.10.4. Переключение между свободно-временным и приближенно-временным моделированием 281
5.11. Мосты транзакций 281
5.12. Интерфейсы DMI и отладки 284
5.13. Комбинированные интерфейсы и сокеты 284
5.14. Пространства имен 285
5.15. Заголовочные файлы и номера версий 285
5.16. Информация о версии программного обеспечения 285
5.17. Примеры использования основных интерфейсов TLM-2.0 286
5.17.1. Транспортные интерфейсы 286
5.17.2. Блокирующий транспортный интерфейс 286
5.17.3. Определение класса блокирующего интерфейса 287
5.17.4. Аргумент шаблона TRANS 287
5.17.5. Правила 287
5.17.6. График последовательности сообщений - блокировка транспорта 288
5.17.6. График последовательности сообщений - временная 289
развязка 289

5.17.7. Схема последовательности сообщений – квантование 290
времени 290
5.17.8. Неблокирующий транспортный интерфейс 291
5.17.9. Определение класса неблокирующего интерфейса 292
5.17.10. Аргументы шаблона TRANS и PHASE 293
5.17.11. Запросы nb_transport_fw и nb_transport_bw 293
5.17.12. Аргумент trans 294
5.17.13. Фазовый аргумент 295
5.17.14. Возвращаемое значение tlm_sync_enum 297
5.17.15. Диаграмма последовательности сообщений – 298
использование пути назад 298
5.17.16. Схема последовательности сообщений - с использованием обратного пути 300
5.17.17. Диаграмма последовательности сообщений – раннее 301
завершение 301
5.17.18. График последовательности сообщений – аннотация 301
времени 301
5.17.19. Аннотации синхронизации с транспортными 302
интерфейсами 302
5.17.20. Примеры стилей кодирования 303
5.18. Интерфейс прямой памяти 306
5.18.1. Определение класса DMI 307
5.19. Примеры программ с использованием TLM-2.0 309
5.19.1. Пример блокирующего интерфейса 309
5.19.2. Пример неблокирующего интерфейса `at_1_phase` 315
5.20. Выводы: Основные характеристики TLM-2 330
Приложение А 332
А.1. Краткое введение в язык С++ 332
А.1.1.Происхождение языка С++ 333
А.1.2. Первая программа в С++ 333
А.2. Переменные С++ 341

А.2.1. Типы данных 341
А.3. Операторы С++ 343
А.4. Конструкции ветвления 345
А.5. Массивы в С++ 351
А.6. Функции в С++ 354
А.7. Указатели в С++. Статические и динамические переменные 357
А.7.1. Пример использования статических переменных 357
А.7.2. Пример использования динамических переменных 358
А.8. Запуск из командной строки 360
А.9. Структуры в С++ 360
А.10. Классы в С++ 362
А.10.1. Модификаторы доступа public и private 366
А.11. Создание объекта через указатель 367
А.13. Конструктор и деструктор класса 368
А.14. Векторы в С++ 372
А.15. Наследование классов в С++ 373
А.16. Дополнительные вопросы по С++ 379
Приложение Б. Установка SystemC-2.3.1 в среде 384
Microsoft Visual Studio 2012 384
Б.1. Предисловие 384
Б.2. Системные требования и указания от Accellera 384
Б.3. Создание библиотек SystemC-2.3.1 385
Б.4. Системные переменные 387
Б.5. Создание SystemC Application 388
Б.6. Установка свойств проекта 390
Б.7. Компиляция и проверка проекта 394
Библиография 396

Глава 1. Проблемы проектирования систем на кристалле

1.1. Введение

Современные тенденции развития заключается в применении встраиваемых систем на основе систем на кристалле (System-on-Chip (SoC)) или (СБИС СнК) в разработке электронных систем. Такие SoC - решения обычно состоят из встроенного процессора (процессоров), встроенных памяти, аппаратных ускорителей (или IP-ядер), высокоскоростных коммуникационных интерфейсов и реконфигурируемой логики. Вследствие этого разработки таких электронных систем становятся все более сложными, поскольку они предъявляют более жесткие требования к более низкой стоимости, более высокой производительности, качеству продукции, безопасности и скорости продаж. Кроме того, в соответствии с законом Мура при дальнейшем развитии возможностей цифрового оборудования, существует потребность, чтобы увеличенное количество функциональных возможностей содержались в более ограниченном пространстве прибора.

Традиционно во всех проектах предусматривалось создание сложной интегральной схемы, состоящей из 500000 вентилей. По существу сложные схемы состояли из основной логики и некоторых жестких макросов, таких как встроенная память. С быстрым прогрессом в технологии обработки полупроводников плотность вентилей на кристалле увеличилась в соответствии с тем, что предсказывал закон Мура. Это помогло реализовать более сложные конструкции на одной и той же ИС.

За последние несколько лет с появлением таких передовых технологий, как услуги мобильной связи, предоставляющие интернет через мобильные телефоны, возрастает потребность в том, чтобы разместить традиционные микропроцессоры, память и периферийные устройства - все на одной микросхеме. Это было отмечено как начало эпохи SoC. Обычно SoC содержит один или более программируемых процессоров, встроенную памяти, таймеры, контроллеры прерываний, шины, специально разработанные сложные аппаратные модули и встроенное программное обеспечение.

Проектирование конструкции системы на кристалле (SoC) имеет отличительные особенности:

• Несколько доменов проектирования: аппаратное, программное, аналоговое;

• Множество исходных компонентов: DSP (Digital signal processor), ASIC (Application-Specific Integrated Circuit), IP (Intellectual Property) -ядра;

• Жесткие ограничения: работа в реальном времени, с низким энергопотреблением.

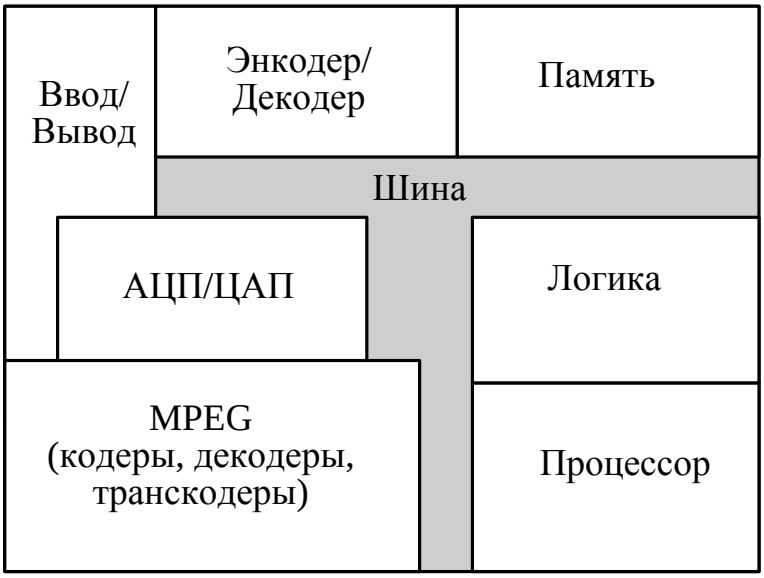


Рис. 1.1. Типичная конструкция SoC

Существенные ресурсы каждого SoC составляют ядра интеллектуальной собственности. Ядро IP – это сложный модуль, выполняющий определенную задачу и созданный для повторного использования. Эти IP - ядра являются строительными блоками для SoC и занимают очень большой процент площади SoC.

Одна из ключевых задач проекта SoC – разбиение системной функциональности по дихотомии на аппаратное обеспечение (HW) и программное обеспечение (SW) или совместно HW/SW. Функциональность, однажды отнесенная к программному обеспечению, теперь для лучшей производительности может быть реализована на аппаратном уровне, а компоненты аппаратного обеспечения должны интегрироваться с программными API (Application Programming Interface) более высокого уровня. В текущей методологии САПР делается априорное разделение и таким образом создаются отдельные аппаратные и программные спецификации. Изменения в разделении HW/SW требуют обширной реорганизации, которая обычно заканчивается неоптимальными проектами. Кроме того, при внедрении встроенных процессоров в ПЛИС, дизайнеры цифровых устройств знакомятся с новой областью САПР, которая включает в себя одновременную разработка как аппаратного, так и программного обеспечения (программа выполненняется на встроенном процессоре).

На рис. 1.2 условно показано соотношение аппаратной и программной части в SoC.

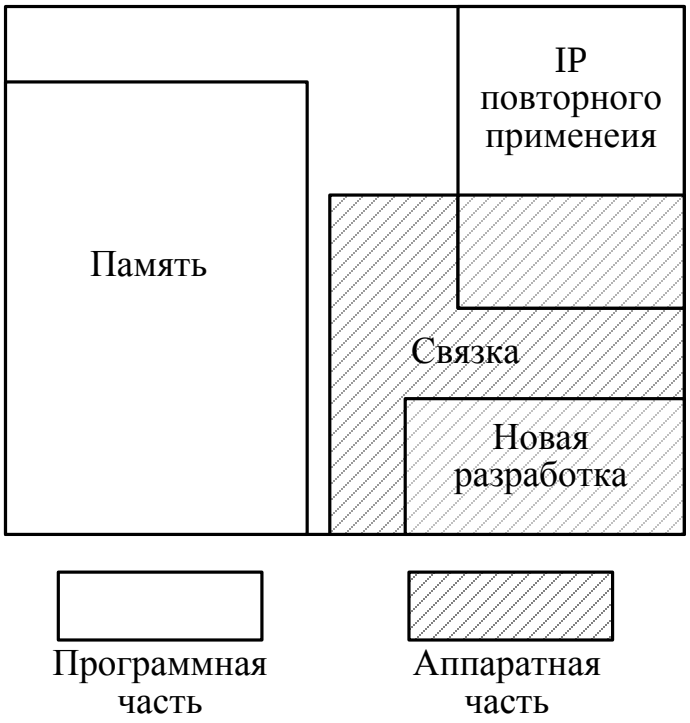


Рис. 1.2. Соотношение аппаратной и программной части в SoC

Еще одним критическим недостатком текущей методологии САПР является то, что она является RTL (register transfer level)-ориентированной и в связи с увеличением сложности схемы время моделирования возрастает и постепенно становится неприемлемым. Понятно, что скорость моделирования полной системы является решающим фактором при разработке сложной цифровой системы, такой как SoC. Эта проблема проверки еще более усугубляется, когда количество тестовых векторов, необходимых для проверки, возрастает в 100 раз каждые шесть лет, что в 10 раз превышает количество вентилей на чипе, указаное законом Мура. Кроме того, полная верификация (проверка соответствия техническим условиям) и валидация (проверка соответствия поставленным функциональным целям) системы часто невозможна до тех пор, пока полностью не построен рабочий прототип. Это особенно актуально для распределенной и гетерогенной среды, такой как сетевая встроенная система.

Очевидно, что для сокращения времени разработки и затрат компьютерные инструменты проектирования (САПР) теперь должны включать функции, которые облегчают проектирование пространственное и архитектурное, а также дают более высокий уровень абстракции. Среди решений вышеупомянутых вопросов проектирования, которые сегодня активно обсуждаются, есть проектирование на уровне абстракции электронной системы (ESL – Electronic System Level) и примение стандартного языка разработки SystemC.

Этот подход потребует совместного проектирования и совместного моделирование аппаратно-программного обеспечения (HW/SW) , он облегчает исследования в области проектирования и дает высокая скорость моделирования. Предложены методологии совместного моделирования, основанные на SystemC, с симулятором набора инструкций в качестве модели процессора в общем исходном файле.

Перечислим основные факторы, которые привели к созданию новой методологии проектирования, основанной на SystemC:

• сложность SoCs продолжает увеличиваться;

• использование моделирования на уровне вентилей или даже RTL для характеристики и изучения полного SoC для типичных сценариев прецедентов не представляется возможным;

• создание моделей на этих уровнях занимает довольно много времени;

• полные модели обычно готовы на поздней стадии разработки системы и внести в них изменения затруднительно или невозможно;

• симуляция идет слишком медленно.

Для решения проблемы требуется:

• модель с более высоким уровнем абстракции на основе SystemC;

• меньше архитектурных деталей в начальном в потоке проектирования;

• более высокая скорость моделирования

• возможность моделирования более сложных систем.

1.2. Цели SystemC

Одной из основных задач SystemC является то, чтобы моделировать на системном уровне. Это моделирование систем над уровнем RTL (register-transfer level) абстракции, в том числе систем, которые могут быть реализованы в программном или аппаратном обеспечении или некоторой комбинации этих составлюющих. Модели RTL имеют целью выполнение системы с использованием регистров и комбинационной логики. Одной из проблем в создании языка проектирования на уровне системы является то, что существует широкий спектр дизайнерских моделей вычисления, дизайна уровней абстракции и дизайна методологии, которые используются в конструкции системы. Для решения в SystemC этой небольшой, но очень важной задачи, к языку было добавлено системное моделирование. На вершине этого языкового фундамента мы можем добавить более конкретные модели вычислений, библиотек проектирования, методических указаний по моделированию и методологии дизайна, которые необходимы для

проектирования системы. Небольшой, универсальный фундамент моделирования в SystemC называется "ядро языка" и является центральным компонентом стандарта SystemC 2.0.

Стандарт SystemC 2.0 был введен OSCI в 2001 году и эта книга использует спецификацию к данному станларту. В настоящее время распространяется версия SystemC-2.3.1, выпущенная в 2014 году, и все примеры взяты нами для этой версии.

Другие компоненты стандарта SystemC 2.0 включают в себя элементарные модели библиотеки, построенные на ядре языка (например, таймеры, FIFOs, сигналы и т.д.), которые широко применяются. Следует признать, что много различных моделей вычислений и проектных методик могут быть использованы в сочетании с SystemC. По этой причине расчетные библиотеки и модели, необходимые для поддержания этих специфических методик проектирования, целесообразно отделять от ядра стандартного языка SystemC 2.0.

SystemC 2.0 - это язык моделирования, основанный на C++. Он расширяет возможности C++, позволяя моделировать описания аппаратных средств, добавляет моделирование аппаратных описаний, библиотеку классов функций, типов данных и другие языковые конструкции на C++. Эта библиотека классов содержит мощные новые механизмы, которые позволяют проектным группам моделировать и проверять конструкции, выраженные в истинных системных уровнях абстракции, уточнить их, чтобы отразить варианты реализации и, наконец, связать модель системы с реализацией и проверкой аппаратного обеспечения.

SystemC является надстройкой C/C++ и содержит специальные библиотеки для моделирования HW.

От C/C++ SystemC включает в себя следующие функции:

• классы и объекты;
• инкапсуляция - данные и поведение;
• перегрузка операторов - новые типы и поведение;
• усиление печати - дополнительная защита;
• наследование - повторное использование декларации;
• шаблоны - шаблоны построения.

Преимущества процесса проектирования на основе C/C++ состоят в следующем:

• производительность;
• технические характеристики между архитекторой и исполнением является изменяемыми;
• высокоскоростное и высокоуровневое моделирование и прототипирование;

• уточнение, отсутствие перевода на аппаратное обеспечение (отсутствие «семантического разрыва»);

• уровень системного моделирования;

• завтрашние разработчики систем будут разрабатывать больше программного обеспечения и меньше аппаратного обеспечения;

• совместная разработка, совместное моделирование, совместная проверка, совместная отладка;

• аспект повторного использования;

• оптимальная поддержка повторного использования объектно-ориентированными методами;

• эффективное повторное использование testbench;

• особенно широко распространен и широко используется C/C++!

Недостатки процесса проектирования на основе C/C++.

• C/C++ не был создан для проектирования аппаратного обеспечения;

• C/C++ не поддерживает:

• аппаратную коммуникацию;

• сигналы, протоколы;

• понятие времени;

• тактирование, операции с чередованием по времени;

• параллельность;

• аппаратное обеспечение является одновременно параллельным, работает параллельно;

• реактивность;

• аппаратура по своей сути реактивна, реагирует на воздействия, взаимодействует со своей средой (требует обработки исключений);

• Типы аппаратных данных;

• тип бита, тип вектора бит, многозначные логические типы, целые типы со знаком и без знака, типы с фиксированной точкой.

SystemC - это единый язык для определения, совместного моделирования, уточнения системы аппаратных и программных компонентов вплоть до уровня передачи регистров для синтеза. SystemC предоставляет ядро для моделирования исполняемой системы, написанное в SystemC. Он может быть использован для описания циклических точных моделей на системном уровне, разработки алгоритмов программного обеспечения и аппаратной архитектуры.

SystemC отвечает следующим требованиям:

• разрешает совместное проектирование аппаратного и программного обеспечения и совместную проверку;

• быстрое моделирование для проверки и оптимизации;

• плавный переход на аппаратное и программное обеспечение;
• поддержка повторного использования проекта и архитектуры;
• тактирование.

Проектирование систем на кристалле (SoC) необходимо выполнять на четырех уровнях абстрактного описания:

архитектура;

поведенческий уровень;

RTL – уровень;

уровень вентилей.

Скорость итераций увеличивается при проектировании на системном уровне.

Рис. 1.3. Изучение конструкций на разных уровнях абстракции

К языку программирования на системном уровне предъявляются следующие требования:

1. Спецификация и дизайн на различных уровнях абстракции;

2. Создание исполняемых спецификаций конструкции;

3. Создание исполняемых платформ моделей, представляющих возможную реализацию архитектуры;

4. Быстрое моделирование для того, чтобы исследовать дизайн-пространство;

5. Конструкции для разделения функциональности от коммуникаций.

Сравнение применимости различных языков проектирования для решения задач разного уровня показано на рис. 1.4. Как видно, SystemC

имеет наиболее широкие возможности моделирования систем на разных уровнях описания.

Сравнение языков

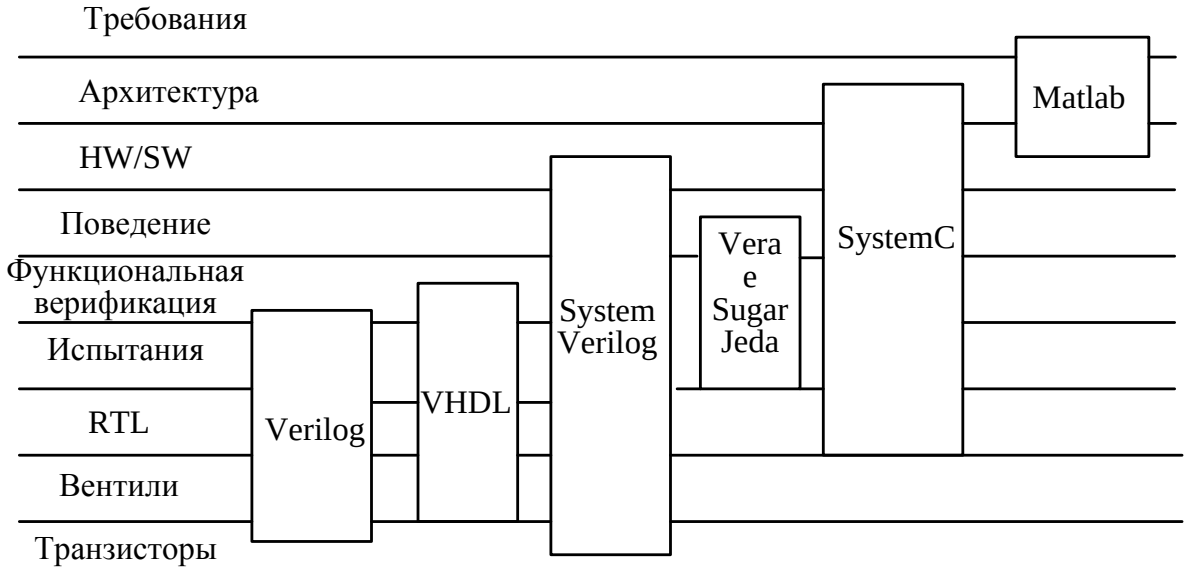


Рис. 1.4. Сравнение применимости различных языков проектирования

В связи с быстро возрастающей сложностью конструкции и ростом стоимости ошибки или отказа, разработчикам системы в большинстве областей продукции требуется похожий подход проектирования сверху вниз, но с улучшенной методологией. Эта возникшая методология основана на моделировании на уровне транзакций (*Transaction – level model* -TLM) и используется в SystemC.

Моделирование на уровне транзакций является новой концепцией без точного определения. Рабочая группа Open SystemC Initiative (OSCI) в настоящее время дала определение набора терминологии для TLM и в конечном итоге развивает TLM стандарты. На самом деле, когда инженеры говорят о TLM, они, вероятно, говорить об одном или нескольких из четырех различных стилей моделирования.

Использование TLM позволяет испытывать модели на ранних этапах процесса разработки системы. TLM является концепцией, которая не зависит от языка. Тем не менее, для реализации и результативности TLM модели, полезно иметь такой язык, как SystemC, который имеет функции поддержки независимого уточнения функциональных возможностей и коммуникаций, что имеет решающее значение для эффективного развития TLM.

1.3. Уровни моделирования в SystemC

В SystemC используют семь уровней моделирования:

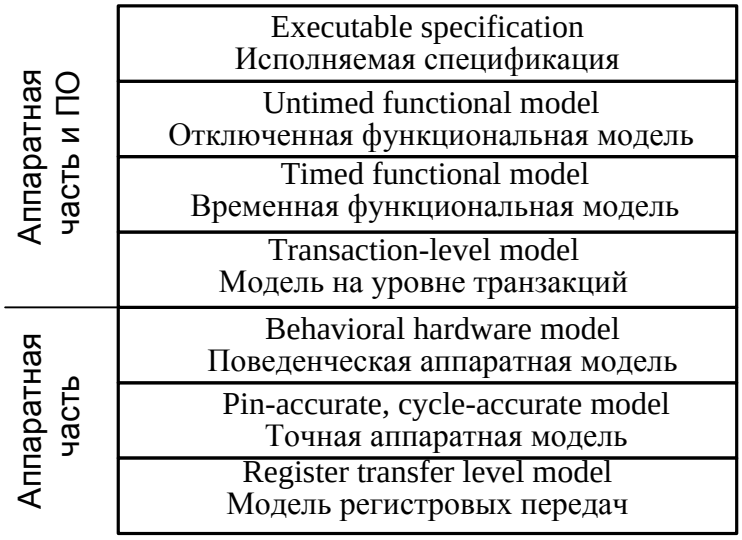


Рис. 1.5. Уровни моделирования в SystemC

1. ***Executable specification*** (исполняемая спецификация) – это модель, которая прямо передает спецификацию проекта в SystemC. Модель предусматривает функционирование конструкции таким образом, который независит от возможного выполнения. Если присутствуют временные задержки, они учитываются в исполняемой модели.

2. ***Untimed functional model*** подобна предыдущей, но временные задержки не присутствуют в модели. Обычно коммуникации моделируются с использованием FIFO с блокировкой записи и чтения, так что данные надежно передаются между моделями.

3. ***Timed functional model*** – по-прежнему прямая передача данных, задержки добавлены к процессам и отражают особенности функционирования модели. Применяют на ранних стадиях компромиссного анализа hardware-software.

4. ***Transaction-level model***

Связи между модулями смоделированы с использованием функций вызовов. В такой модели коммуникации обычно моделируются путем, который точен в терминах функционирования и часто в терминах времени, но коммуникации не моделируются структурно точно. Например, в TLM для SoC платформы мы можем моделировать различные типы транзакций, которые поддерживаются на шине чипа (короткие транзакции чтение/запись), но мы не можем моделировать реальные проводники шины или пины, которые соединяют модуль с шиной.

Когда используют термин ***platform transaction-level model***, показывают, что модель использует TLM стиль с целью моделировать инфраструктурные связи в платформе SoC и что модули вне такой конструкции

структурно соответствуют блокам вне целевой реализации. Этот метод используют для уточнения эффектов в модели, и он является более качественным, точным и эффективным путем моделирования взаимосвязи HW и SW на самом раннем этапе проектирования.

5. Behavioral hardware model – это модель, имеющая контактную и функциональную точность на границах, тактовая точность на границах не учитывается. Эта модель может использоваться как входная для средств поведенческого синтеза.

6. ***Pin-accurate, cycle-accurate hardware model*** – это модель обеспечивает контактную и тактовую точность на границах в добавление к функциональной точности. Для модели не требуется внутренняя структура, которая влияет на целевую реализацию.

7. ***Register - transfer level model*** имеет на своих границах контактную и тактовую точность. В добавление внутренняя структура RTL модели точно отражает регистры и комбинационную логику целевой реализации.

1.4. Краткая история создания и развития SystemC

SystemC является результатом эволюции многих концепций в исследовательских и коммерческих сообществах EDA (Electronic Design Automation). Многие исследовательские группы и EDA компании внесли свой вклад в этот язык.

SystemC начиналась как очень ограниченный циклический симулятор и «слегка отличный» от языка RTL. Язык эволюционировал и развивался до истинного языка проектирования системы, который включает как программные, так и аппаратные концепции. Хотя SystemC специально не поддерживает аналоговое оборудование или механические компоненты, нет никаких причин, почему эти аспекты системы не могут быть смоделированы с помощью SystemC-конструкций или методами co-simulation. В последние году выпущена расширенная версия языка SystemC-AMS для проектирования аналоговых и смешанных систем на кристалле.

Некоторые из организаций, которые внесли значительный вклад в изучение языка разработки очень рано поняли, что любой новый язык дизайна должен быть открытыми для сообщества и не быть частным или собственным.

В результате, Open SystemC Initiative (OSCI) была создана в 1999 году. OSCI была сформирована для чтобы:

развивать и стандартизировать язык;

облегчить общение между пользователями и поставщиками языка;

улучшить адаптацию;

обеспечить механизмы для разработки программ с открытым исходным кодом и их поддержание.

Основные даты в развитии SystemC

• Открытая инициатива SystemC (OSCI) - платформа языка и моделирования на основе C++

• Сентябрь 1999 - SystemC 0.9 - первая версия, основанная на цикле.

• Март 2000 - SystemC 1.0 - широко доступный основной выпуск, набор конструкций для RTL и поведенческого моделирования.

• Август 2002 - SystemC 2.0 - каналы и события, более чистый синтаксис, предоставление возможности моделирования на системном уровне для программных и аппаратных реализаций

• Апрель 2005 г. - SystemC TLM 1.0 (моделирование уровня транзакций)

• Сентябрь 2005 г. - SystemC 2.1

• Июль 2006 г. - SystemC 2.2 (обновлено в марте 2007 г.)

• Июнь 2008 г. - SystemC TLM 2.0.0 (библиотека)

• Июль 2009 г. - LRM SystemC TLM-2.0 (библиотека TLM-2.0.1)

• Март 2010 г. - SystemC AMS 1.0 LRM

• Ноябрь 2011 г. - стандарт IEEE 1666-2011

• Июль 2012 г. - SystemC 2.3 (интегрировано с TLM)

• Март 2013 г. - завершена разработка SystemC AMS 2.0

• Апрель 2014 г. - SystemC 2.3.1 (интегрировано с TLM)

SystemC состоит из языковых и потенциально-методологических библиотек. Создана библиотека верификации SystemC. Авторы рассматривают SystemC Verification (SCV) как наиболее значимую из этих библиотек. Эта библиотека добавляет поддержку современного языка проверки высокого уровня и такие концепции, как ограниченная рандомизация, самоанализ и запись транзакций. Первый выпуск библиотеки SCV произошел в декабре 2003 после более чем года бета-тестирования.

Несмотря на то, что цифровые схемы являются специальным типом аналоговых схем и методов, для анализа и понимания аналоговых и смешанных сигналов (AMS) до настоящего времени для проектировщиков аналоговой системы использовался слишком сложный и отнимающий много времени аппарат. Расширение SystemC-AMS для C++ - важнейшая отправная точка в решении этого вопроса и основывается на существующих версиях SystemC. Для удовлетворения особых требований аналоговых систем и цифрового оборудования с программным обеспечением требуется взаимодействие с их физической аналоговой средой. Например, когда цифровое аппаратное / программное обеспечение взаимодействует с радиочастотными системами, датчиками и

исполнительными механизмами и силовой электроникой, анализ должен не только решать проблемы, характерные для чисто аналоговых и чисто цифровых систем, а также моделировать их взаимодействие в режиме реального времени. Этим специальным требованиям отвечает SystemC-AMS.

Для изучения книг по SystemC требуется, чтобы читатель владел практическими знаниями C++ и минимальными знаниями аппаратного обеспечения. Для навыков C++ не требуется, чтобы, читатель являлся «мастером». Надо, чтобы Вы хорошо знали синтаксис, объектно-ориентированные функции и методы использования C++.

Краткие сведения о языке C++ и объектно-ориентированном программировании приведены в приложении к этой книге.

Чтобы полностью понять примеры, читателю потребуется минимальное понимание цифровой электроники.

1.5. Методология проектирования SystemC

Чтобы понять методологию проектирования SystemC и ее преимущества, важно сначала понять методологию, отличную от SystemC.

В методологии non-SystemC (рис. 1.6) разработчики системы должны написать исполняемые спецификации в C или C++, а затем проверить и отладить этот проект. Обнаружив, что эта конструкция удовлетворяет всем техническим требованиям, она передается в группы разработки RTL. Затем группа проекта RTL перезаписывает проект в RTL, чтобы синтезировать его для вентилей. В такой методологии функциональное RTL-описание иногда зависит от выполняемых спецификаций и, следовательно, становится склонным к ошибке. Кроме того, возникает реальная проблема, если на уровне RTL обнаруживается, что что-то в концептуальной модели не может быть реализовано, так как нет общей среды проектирования между разработкой системы и ее реализацией.

В методологии SystemC разработчику системы нужно только написать модель SystemC. Разработчик может итеративно уточнять исполняемые спецификации вплоть до уровня регистровой передачи, который все еще находится в SystemC до синтеза. Испытательные программы (Testbench) можно использовать повторно, чтобы гарантировать, что итерационный процесс не имеет ошибок. Если во время реализации RTL обнаружено, что что-то концептуально неправильно, гораздо проще его переписать.

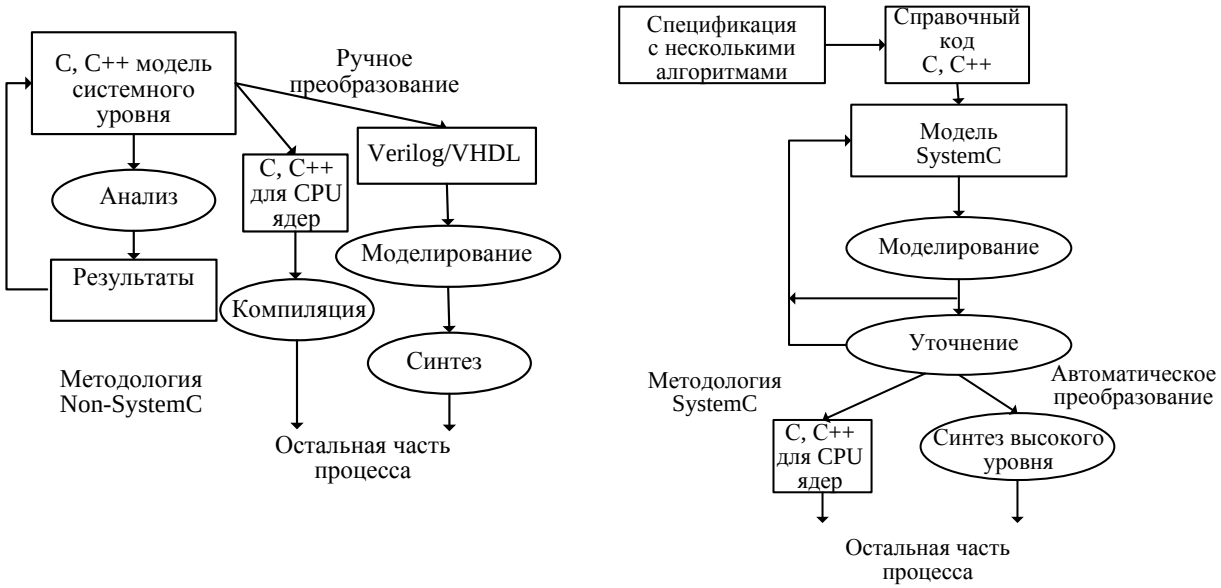


Рис. 1.6. Сравнение методологии проектирования

SystemC использует следующие типы моделей:

- архитектурная модель системы;
- модель производительности системы;
- модель уровня транзакций (TLM);
- функциональная модель;
- модель передачи на уровне регистров (RTL).

Некоторые проекты SystemC начинаются как функциональные модели, в то время как большая часть кода SystemC делается на уровене TLM. Почти всегда TLM выступает в качестве исполняемой платформы, которая является достаточно точной для запуска программного обеспечения.

Главной причиной использования SystemC является значительное увеличение производительности симуляции на уровне TLM по сравнению с исполняемыми платформами, смоделированными на уровне RT с использованием Verilog или VHDL. Модели SystemC TL достаточно быстры, чтобы служить платформой для разработки программного обеспечения, что позволяет выполнить раннюю разработку программного обеспечения и совместное моделирование аппаратного и программного обеспечения. Обе TL и функциональные модели достаточно быстры для архитектурного моделирования и анализа на системном уровне.

Типичные числа для модели System on Chip составляют от ~ 1K циклов в секунду до более высоких ~ 300-400K циклов в секунду. Этот диапазон зависит от того, как процессор моделируется в системе. Если использован симулятор набора инструкций (ISS), то производительность обычно составляет от 1 до 10 тыс. циклов. Если процессор моделируется

как прямое подключение к системной шине, то производительность переходит в диапазон 100K и выше.

Вторая причина использования SystemC - функциональная проверка. Одна и та же исполняемая платформа, которая используется для разработки программного обеспечения, часто используется и для проверки всей системы. Эта проверка происходит на раннем этапе проекта, и TLM становится прекрасной проверкой для всей системы.

Поскольку SystemC - это C++, у него есть ряд неотъемлемых свойств, таких как классы, шаблоны и наследование, которые поддаются проверке. Эти возможности дополняются SystemC Verification Library (SCV), что делает SystemC мощным языком проверки, а также языком моделирования.

Ключевые характеристики SystemC:

- параллельность – синхронные и асинхронные процессы;
- понятие времени - несколько тактовых генераторов с произвольным фазовым соотношением;
- типы данных - битовые векторы, целые числа с произвольной точностью;
- типы данных фиксированной точки произвольной точности;
- связь - сигналы, каналы;
- расширенные протоколы связи;
- реактивность - просмотр событий;
- поддержка отладки - вывод осциллограмм;
- поддержка моделирования;
- поддержка множества уровней абстракции и итеративной обработки;
- поддержка создания функциональной модели.

Потоки SystemC и использование

На рис. 1.7 показан типичный системный поток SystemC. TLM часто служит как исполняемая платформа для программного обеспечения и как ценная контрольная модель для проверки. Путь к вентилям от TLM либо прямой с использованием поведенческого синтеза, либо TLM служит для ручного перевода либо в SystemC на уровень RTL-модели, либо в HDL (Verilog или VHDL) RTL-модель с последующим синтезом.

Верификация выполняется на уровне TLM. Поскольку TLM переводится на уровни RTL или вентилей (gates), та же самая проверка может быть использована с помощью адаптеров (иногда называемых транзакторами).

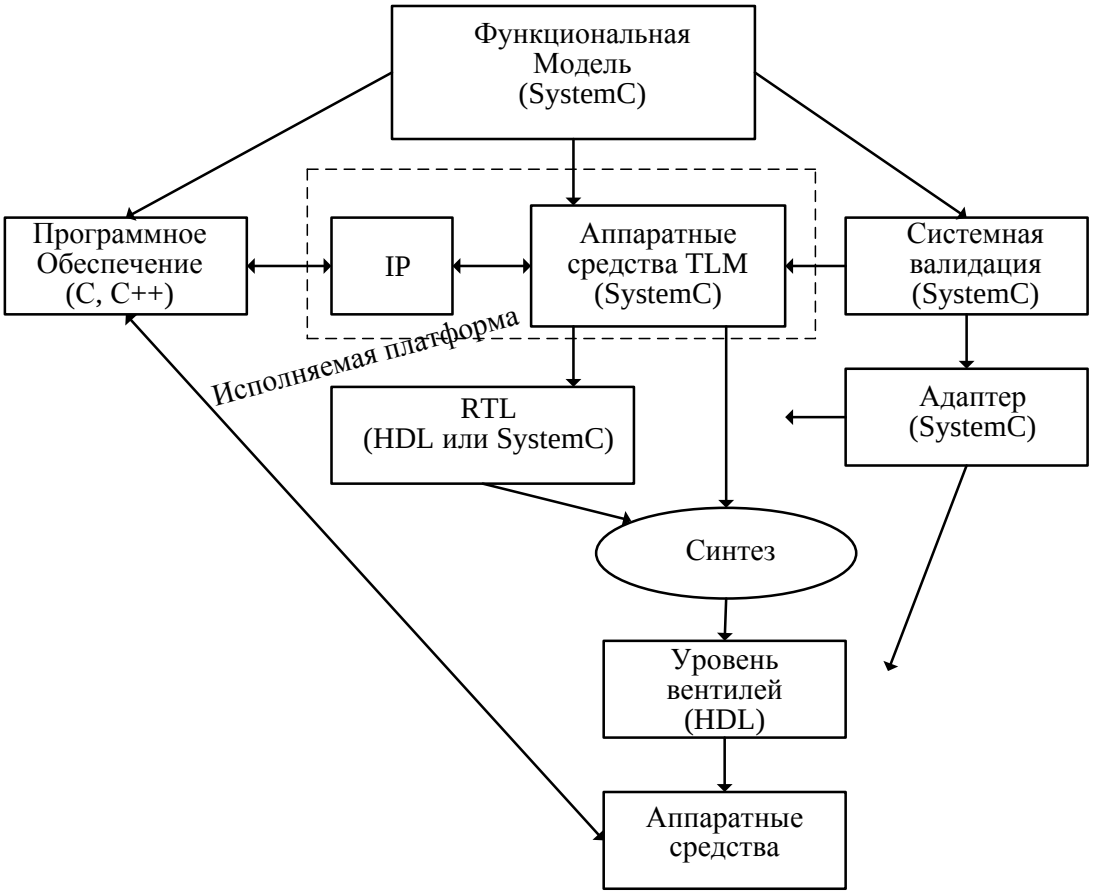


Рис. 1.7. Поток системного проектирования SystemC

Модели вычислений

Наиболее известные модели вычислений в SystemC включают следующие:

статический многоскоростной поток – включает три фазы: чтение входных данных, выполнение вычисление внутри процесса, вывод всех данных, количество всех данных является известным и неизменным;

динамический многоскоростной поток;

сетевой процесс Кана – эффективная модель вычислений для создания алгоритмических моделей в применении к сигнальным процессорам, в которых процессы выполняются конкурентно, и каналы FIFO имеют определенную длину.

Дискретные события, которые используют:

RTL (register transfer level) моделирование HW-связано с цифровой аппаратурой, синхронизированной во времени;

Сетевое моделирование (стохастические или ожидающие модели);

TLM (transaction-level model) - платформа моделирования SoC , основанная на транзакиях, мы моделируем коммуникации между модулями, используя функции вызова, которые представляют транзакции, типично поддерживаемые целевой платформой. Каналы sc_signal используются вместо данных. Эти сигналы обменяются между различными процессами, путем чтения и записи общих переменных данных.

TLM проекты более краткие и быстрые, чем соответствующие RTL проекты.

Перечислим основные уровни абстракции и модели использования SystemC.

Сюда входят:

– функциональное моделирование системных алгоритмов;

– моделирование системных архитектур на уровне транзакций;

– моделирование на уровне RTL и привязка SystemC к маршрутам реализации;

– недавние добавления к SystemC на тему верификации: библиотека SCV верификации SystemC.

SystemC является языком, идеально подходящим для моделирования встроенных систем, и именно тем языком, с помощью которого можно не только описывать сами модели, но и разрабатывать тестовое окружение для этих моделей и использовать его для тестирования реальных плат.

1.6. Стандартные графические обозначения

Для иллюстрации моделей в SystemC применяют стандартные графические обозначения, как показанные на рис. 1.8. Терминалогия их будет представлена по мере изучения книги.

SystemC использует соглашение об именах, где большинство идентификаторов SystemC имеет префикс sc_ или SC_. Это соглашение зарезервировано для библиотеки SystemC и Вы не должны использовать его в коде конечного пользователя.

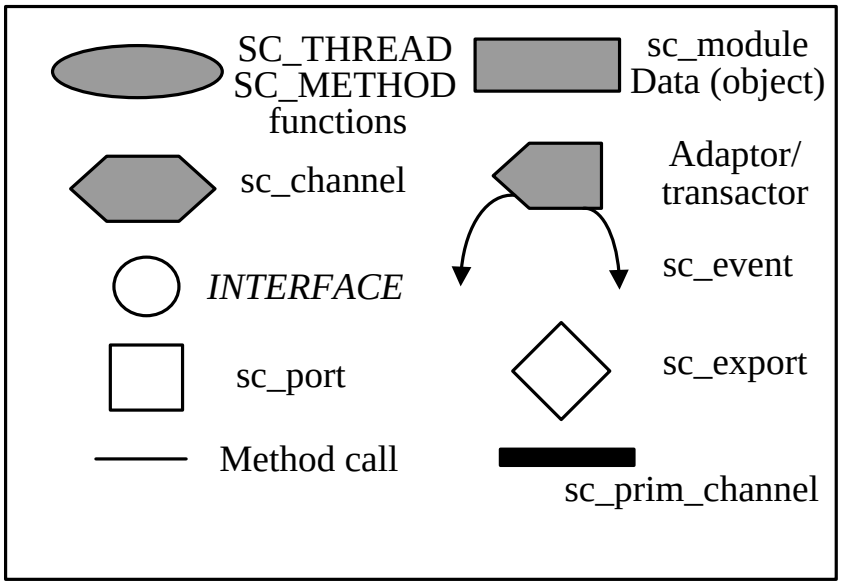


Рис. 1.8. Графические обозначения в SystemC

1.7. Как изучать эту книгу

Успешно изучить язык программирования Вы сможете только, постоянно подкрепляя новые теоретические сведения самостоятельной практической работой на компьютере. Полезные фрагменты листингов программ будут лучше запоминаться, поиск и устранение ошибок даст Вам навыки работы в интерактивных средах программирования, которые показывают ошибки и дают подсказки, как их исправить.

Поэтому эта книга иллюстрирует теорию множеством практических примеров листингов программ. Причем, почти все примеры проверены автором моделированием в средах Eclipse или Microsoft Visual Studio. Скриншоты результатов моделирования подтверждают это. Большинство примеров были найдены в руководствах по SystemC от разработчиков из OSCI и разработчиков ASIC (Application-Specific Integrated Circuit - интегральные схемы специального назначения). Многие материалы, найденные мною в Интернете, я перевел на русский и отредактировал. Полагая, что читетели владеют техническим английским языком, я не стал переводить на русский некоторые комментарии в программах и общепринятые термины.

Выполняя важное правило, по которому, приступая к сложной работе надо подготовить хорошие инструменты и научиться пользоваться ими, во второй главе книги Вы научитесь устанавливать на своих компьютерах программные средства: компилятор Cygwin, библиотеки SystemC, среду разработки Eclipce, программу GTKWave для отображения результатов моделирования. В Приложении Б показано как установить и настроить для SystemC среду разработки Microsoft Visual Studio.

В третьей главе Вы изучите основы языка SystemC-2.3.1. Все теоретические сведения подкрепляются практическими примерами программ и я рекомендую Вам самостоятельно выполнить моделирование и добиться совпадения результатов с приведенными в книге.

Четвертая глава содержит более сложные практические примеры моделирования, в которых в комплексе применены сведения о SystemC, полученные в третьей главе.

Пятая глава посвящена моделированию на уровне транзакций с использованием стандарта TLM-2.0. Этот метод применяют для разработки на системном уровне с исследованием передач между моделями и объектами сложной системы.

Я надеюсь, что эта книга станет для Вас первым шагом в освоении сложной современной технологии проектирования «Систем на кристалле».

Желаю Вам успехов !
Профессор В.А. Алехин

Глава 2. Установка SystemC в интерактивных средах разработки Eclipse и Micrisoft Visual Studio

Как мы отмечали ранее, SystemC создан на основе языка C++. Проект на языке SystemC состоит из набора файлов .cpp и .h. Для того чтобы полноценно работать, не требуется интерактивная среда разработки, поддерживающая данный язык. Достаточным условием является наличие компилятора C++ и библиотеки SystemC. На рис. 2.1 изображена схема процесса компиляции проекта на SystemC.

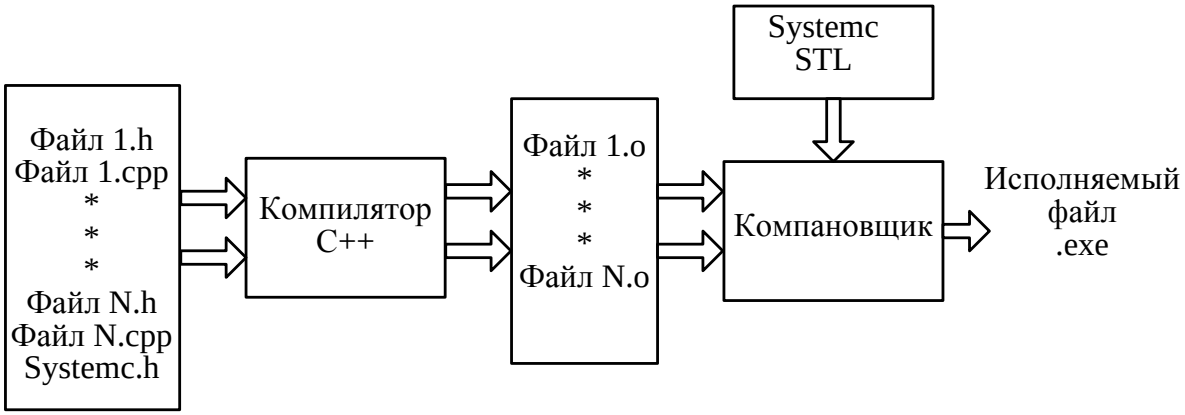


Рис.2.1. Процесс компиляции проекта SystemC

Компилятор C++ обрабатывает файлы .h и .cpp из набора файлов проекта SystemC и создает объектные файлы (расширение файла .o). После создания объектных файлов компоновщик связывает созданные объектные файлы и соответствующие объектные файлы библиотеки SystemC (и других библиотек, таких как стандартная библиотека шаблонов или STL). В результате создаётся исполняемый файл, который содержит в себе ядро моделирования SystemC с функциональными возможностями исходного проекта.

Компилятор и компоновщик должны располагать информацией о местонахождении файла заголовка (systemc.h) и откомпилированной библиотеки SystemC.

Есть такое правило: «Перед началом серьезной работы приготовь хорошие инструменты !». Проектирование систем в SystemC – это сложная и кропотливая работа. Поэтому мы выбираем наиболее эффективные среды разработки Eclipse и Microsoft Visual Studio.

Установка сред разработки и знакомство с ними позволит Вам глубже изучить последующие разделы и самостоятельно моделировать и проверять работу программ, листинги которых приведены в книге.

В Интернете можно найти много статей и обсуждений на форумах, которые содержат разные рекомендации по установке SystemC в этих

средах. К сожалению, интерфейсы сред нередко меняются в каждой новой версии. Разработчики SystemC, выпуская новую версию, также изменяют указания по установке. В результате автору пришлось потратить достаточно много времени, чтобы подготовить эту главу, экспериментально исправив ошибки в чужих рекомендациях и многократно проверив надежность работы изложенных ниже методик установки SystemC.

Важное уточнение сред, компиляторов и дополнительных программ, использованных в этой книге:

Среды разработки:

Eclipse IDE for C/C++ Developers ,Version: Neon Release (4.6.0)
Microsoft Visual Studio Ultimate 2012 version 11.0
Eclipse в среде Ubuntu

Компиляторы:

Cygwin
Cygwin 64
Linux

Программа вывода временных диаграмм

GTK Wave

2.1. Установка SystemC в Eclipse с компилятором Cygwin

Перед установкой SystemC требуется установить необходимые дополнительные инструменты, а именно, компилятор Cygwin.

Рассмотрим последовательность установок для будущей работы в среде Eclipse.

2.1.1. Краткие сведения об Eclipse

Eclipse (в переводе с английского — «затмение») — свободная интегрированная среда разработки (IDE- Integrated Development Environment) модульных кроссплатформенных приложений. Развивается и поддерживается Eclipse Foundation.

Наиболее известные приложения на основе Eclipse Platform — различные «Eclipse IDE» для разработки ПО на множестве языков (например, наиболее популярные Eclipse JDT (Java Development Tools) для языка Java и Eclipse CDT (C/C++ Development Tools), который служит для разработок на языках C/C++ и используется, в частности, для SystemC.

Мы будем применять платформу Eclipse CDT для разработки проектов на SystemC (надстройке для языка C++).

Eclipse CDT имеет встроенные компиляторы, однако для работы с SystemC рекомендуется перед установкой и запуском Eclipse установить наиболее мощный компилятор Cygwin.

2.1.2. Cygwin

Cygwin (произносится - sigwin) — UNIX-подобная среда и интерфейс командной строки для Microsoft Windows. UNIX – это семейство переносимых, многозадачных и многопользовательских операционных систем. Cygwin обеспечивает тесную интеграцию приложений, данных и ресурсов Windows с приложениями, данными и ресурсами UNIX-подобной среды. Из среды Cygwin можно запускать обычные приложения Windows, также можно использовать инструменты Cygwin из Windows. Это свободное ПО, опубликованное под GNU General Public License версии 2.

2.1.3. Установка Cygwin

Программный файл "setup.exe" можно загрузить с сайта www.cygwin.com. Мы загрузили одну из последних версий Cygwin.

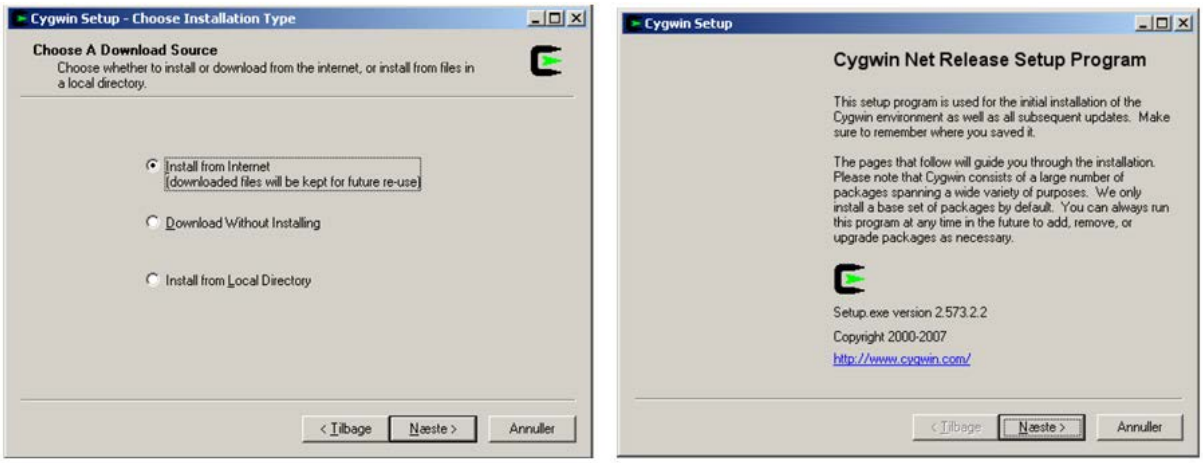


Рис. 2.2. Запуск установки Cygwin

В окне выбора загрузки выбираем Install from Internet (рис. 2.2).

Выбор директории не рекомендуется делать в Program Files, поэтому создаем директорию на диске C. Для удобства переноса всего комплекта установленных для SystemC программ с одного компьютера на другой мы создали директории C:/SystemC/ и в ней поддиректории …/cygwin/ и …/systemc231 (рис. 2.3). Тогда на другом компьютере потребуется только установить «Переменные среды» и «Системные переменные». В этом же окне важно выбрать “Install for all users” и “DOS/text”. Выбор “DOS/text” очень важен для интеграции с Xilinx EDK.

Если Вы не планируете использовать установочный комплект SystemC на других компьютерах, устанавливайте Cygwin в директорию C:/Cygwin.

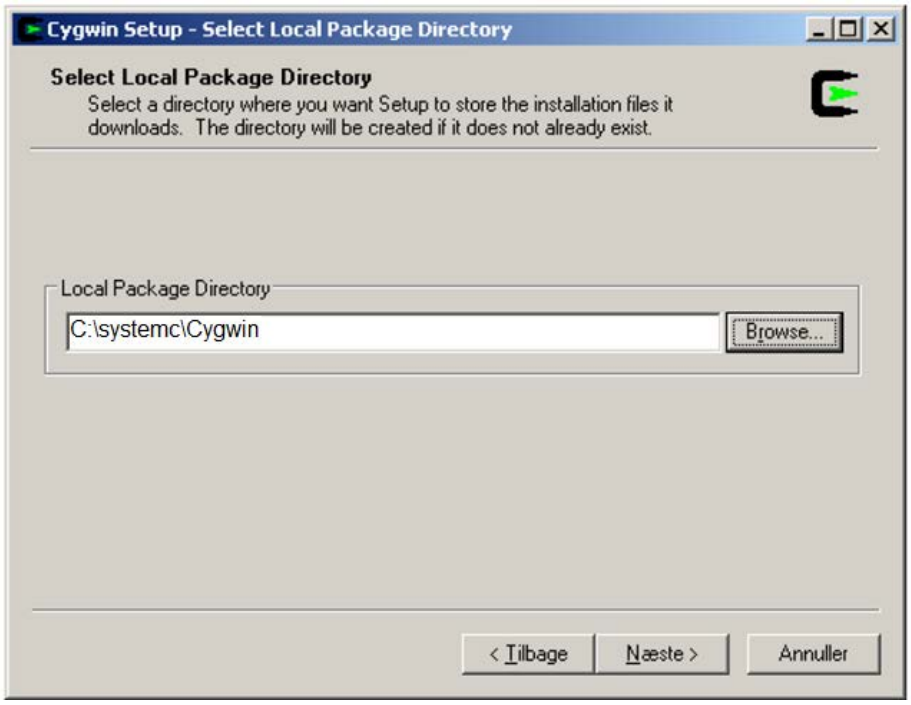


Рис. 2.3. Выбор директории

На следующем шаге надо специфицировать локальную директорию, в которой будут находиться скачанные из Интернета файлы. Если директория еще не существует, она будет создана.

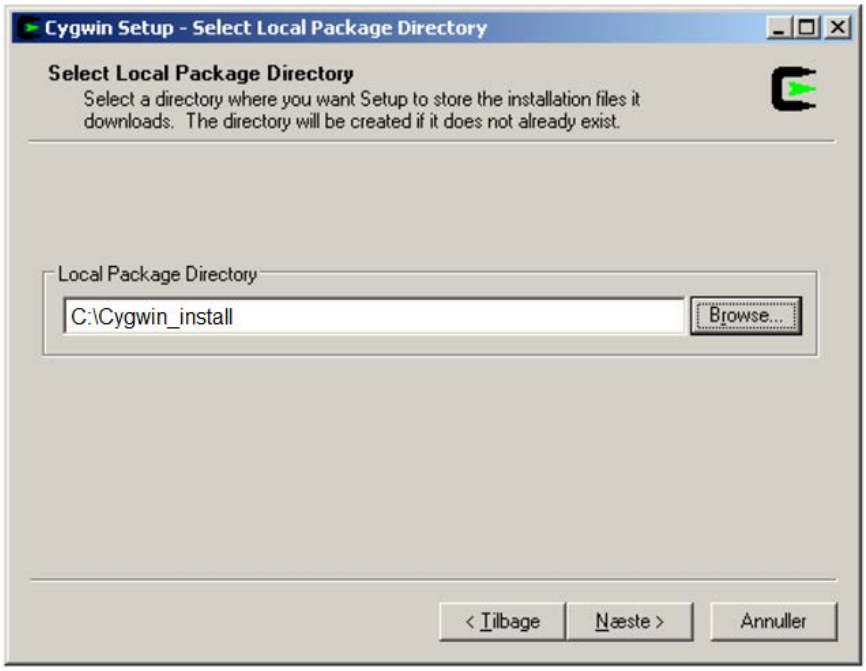


Рис. 2.4. Выбор директории для загруженных файлов

Если Вы выбрали связь через Интернет, тогда включайте Direct Connection или Use IE5 Setting и определите зеркальный сайт. Возможно это будет http://cygwin.com/mirrors.html

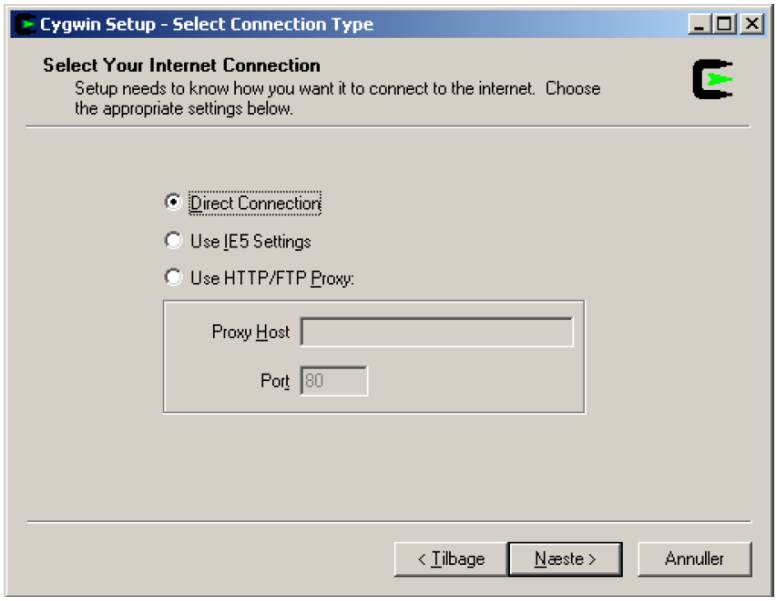


Рис. 2.5. Выбор соединения

Следующая страница показывает группы категорий, которые можно установить. Выберите Base, Devel, Editors, Libs (рис. 2.6). Отметим, что полный пакет Cygwin занимает более 10 Гбайт. Поэтому не следует устанавливать лишнее.

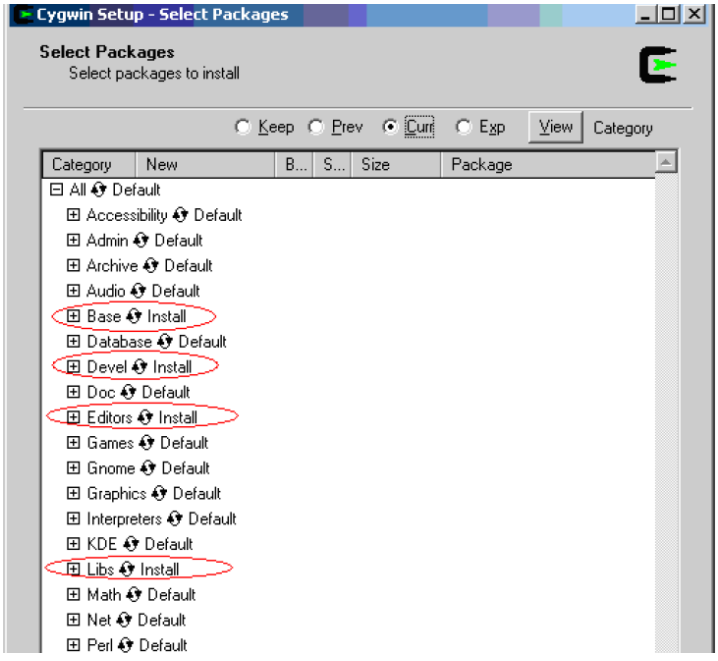


Рис. 2.6. Выбор категорий установки

После окончания установки нажмите Finish.

Установленная программа имеет следующую структуру директории (рис. 2.7):

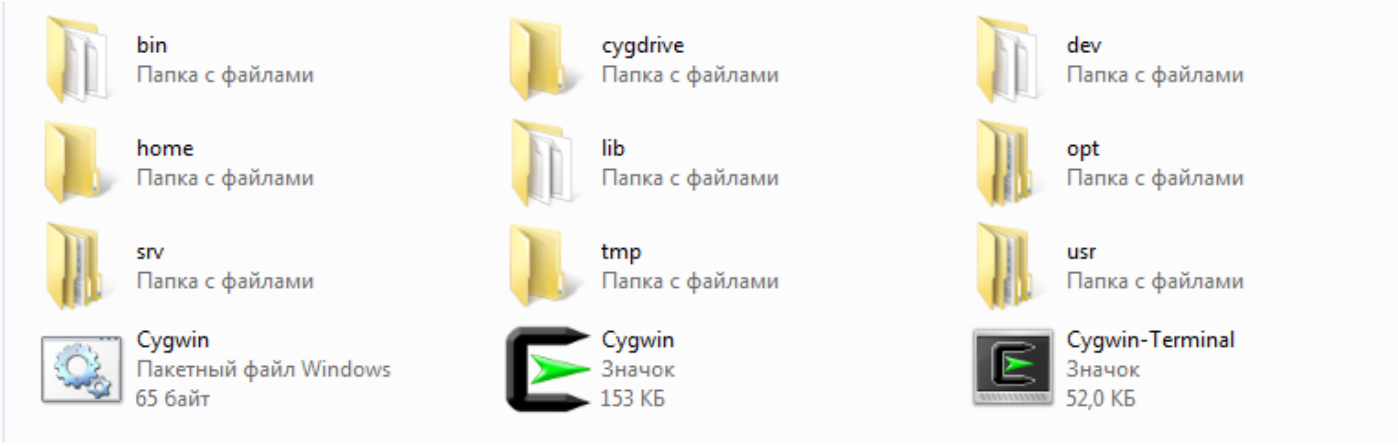


Рис. 2.7. Содержание директории Cygwin

После этого Вы должны добавить C:\Cygwin\bin в путь переменной среды или **скопировать файл cygwin1.dll в директорию C:\Windows\System32.** Это требуется для того, чтобы запускаемые программы компилировались и компоновались Cygwin.

Директорию Cygwin Binary надо включить **в переменные среды** (рис. 2.8).

Для этого в Windows выполняем:

Панель управления – Система - Дополнительные параметры - Переменные среды - Системные переменные.

Задаем:

Переменные среды:

CYGWIN_HOME =C:\ SystemC\cygwin\bin

PATH = C:\ SystemC\cygwin\bin

Системные переменные:

PATH = C:\ SystemC\cygwin\bin

CYGWIN_HOME = C:\ SystemC\cygwin\bin

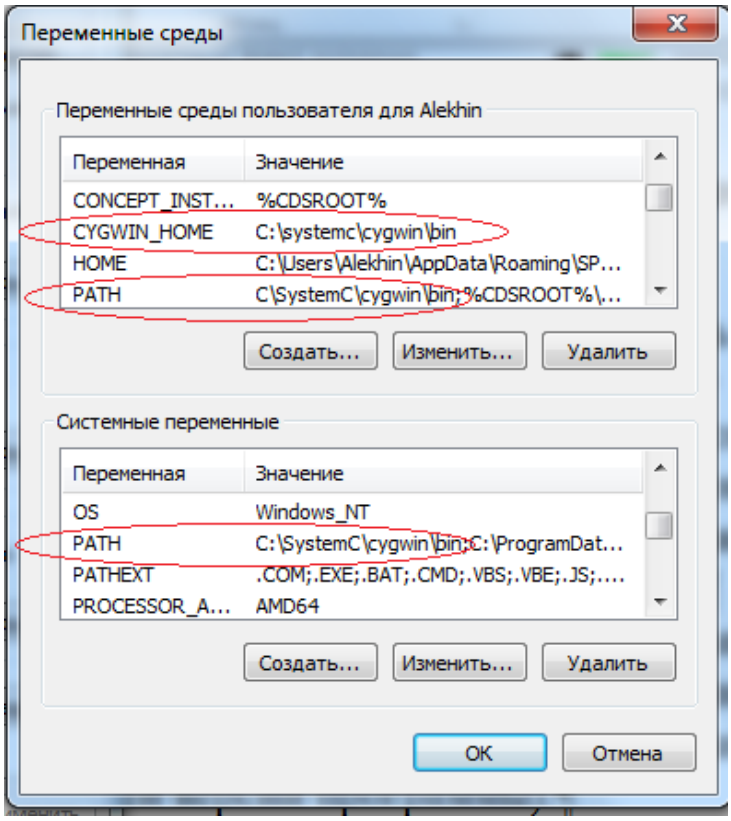


Рис. 2.8. Переменные среды и системные переменные для Cygwin

Для того, чтобы гарантировать, что инструменты доступны, сначала убедитесь, что ваша переменная "путь" включает в себя путь в Cygwin директории bin :C:\SystemC\Cygwin \ bin.

Проверьте путь, введя следующую команду в Windows окне командной строки: gcc –v (рис. 2.9).

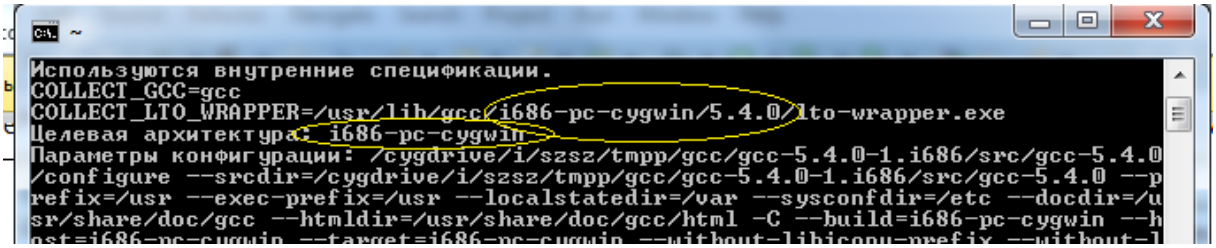


Рис. 2.9. Проверка пути к Cygwin

Вторая команда для проверки make –v (рис.2.10).

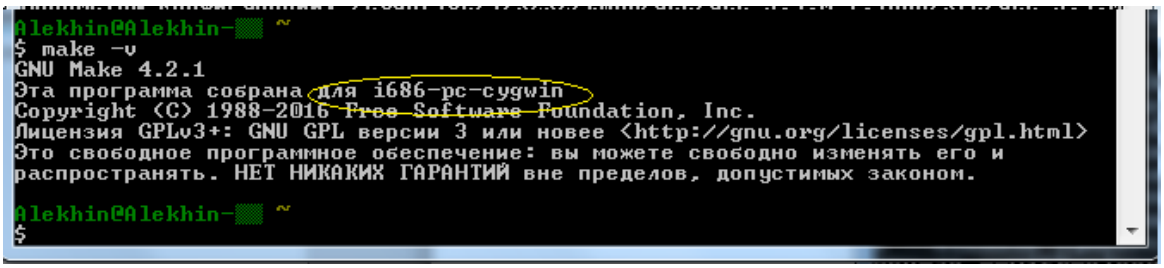


Рис. 2.10. Вторая проверка установки Cygwin

2.1.4. Загрузка программы SystemC-2.3.1

С сайта компании Accellera http://www.accellera.org загружаем архив файлов программы SystemC-2.3.1. После распаковки архива, выполненной два раза, получим следующие файлы программы (рис.2.11):

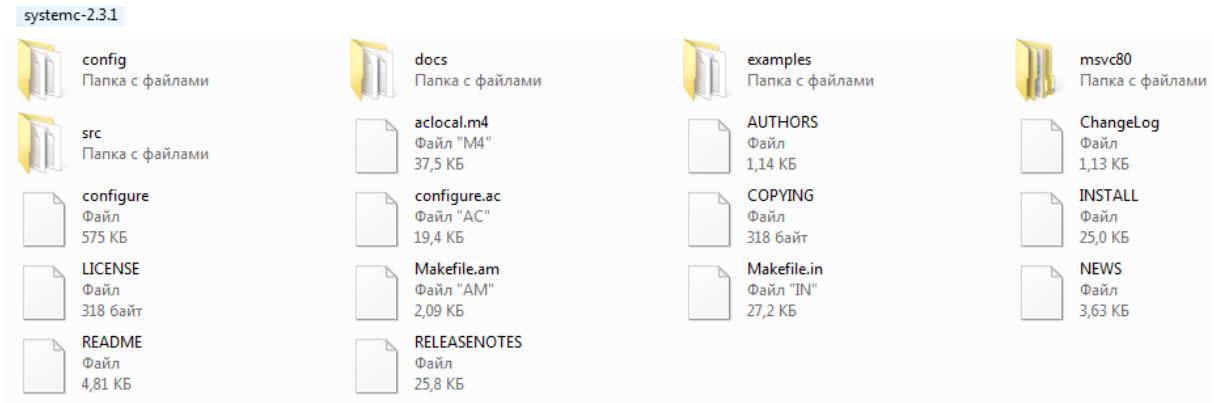


Рис. 2.11. Файлы SystemC-2.3.1 до компиляции

2.1.5 Компиляция SystemC-2.3.1 в Cigwin64

Программа SystemC-2.3.1 у нас установлена в директории C:/systemc/systemc2.3.1/.

Для создания библиотек SystemC требуется выполнить компиляцию с помощью Cygwin.

Запускаем Cygwin и открываем окно **bash** (стандартный командный интерпретатор линукс - GNU **Bourne-Again SHell**). Для этого в командной строке Windows вводим: C:\systemc\cygwin\cygwin.bat (Рис. 2.12). Терминал можно запустить, щелкнув на пакетном файле Cygwin в директории компилятора (рис. 2.7).

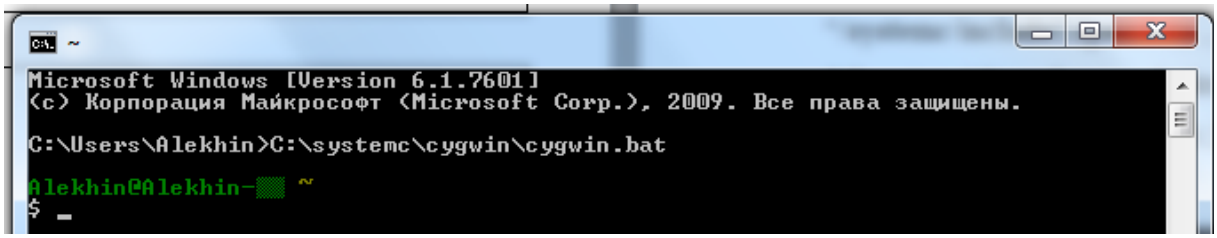


Рис. 2.12. Открытие терминала Cygwin

Затем надо выполнить:

1. Перейти в директорию SystemC (*cd /systemc/).*
2. Создать папку obj dir (*mkdir objdir)* и перейти в неё *(cd objdir).*
3. Запустить конфигурацию пакета (*../configure*) (рис 2.13).
4. Выполняем команду (*make*).

5. Выполняем команду (*make install*).

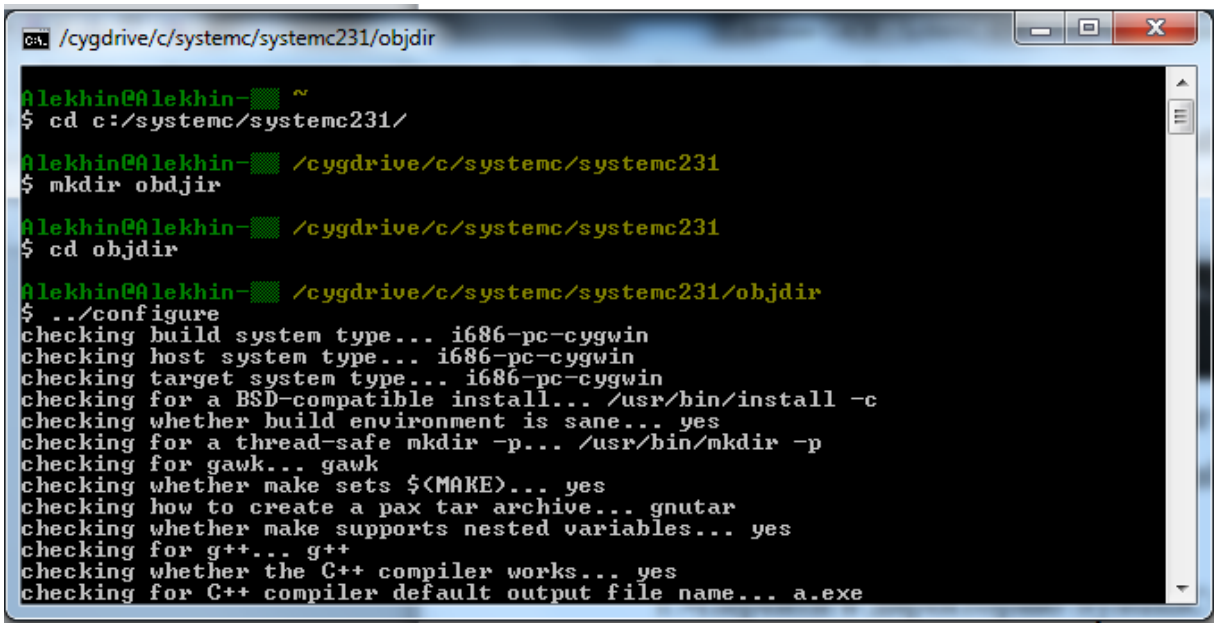


Рис. 2.13. Процесс компиляции библиотек SystemC

Команды *configure*, *make*, *make install* выполняется достаточно долго и сопровождаются выводом различной информации о процессе компиляции (рис. 2.14).

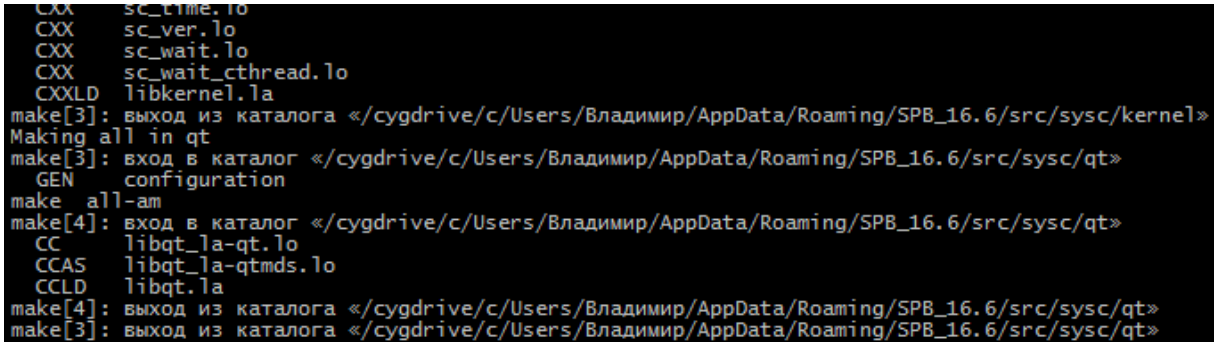


Рис. 2.14. Информация о компиляции

По окончании компиляции появятся три папки “/systemc/include/”, “/systemc/lib-cygwin”, и “/systemc/obdjir/”необходимые для подключения к проектам (рис. 2.15).

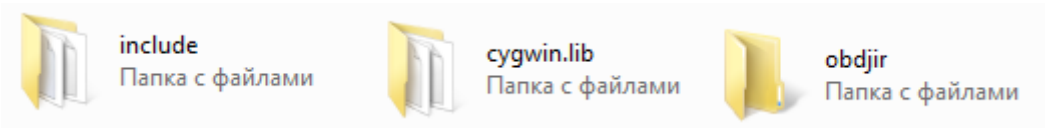


Рис. 2.15. Новые папки после компиляции

2.1.6. Создание переменных сред для SystemC

Устанавливаем для SystemC переменные среды (рис. 2.16):
SYSTEMC=C:\systemc\systemc231\msvc80
SYSTEMC_HOME =/cygdrive/c/systemc/cygwin.
Устанавливаем системную переменную:
SYSTEMC=C:\systemc\systemc231\msvc80\SystemC.

Добавляем путь:
PATH=C:\SystemC\cygwin\bin

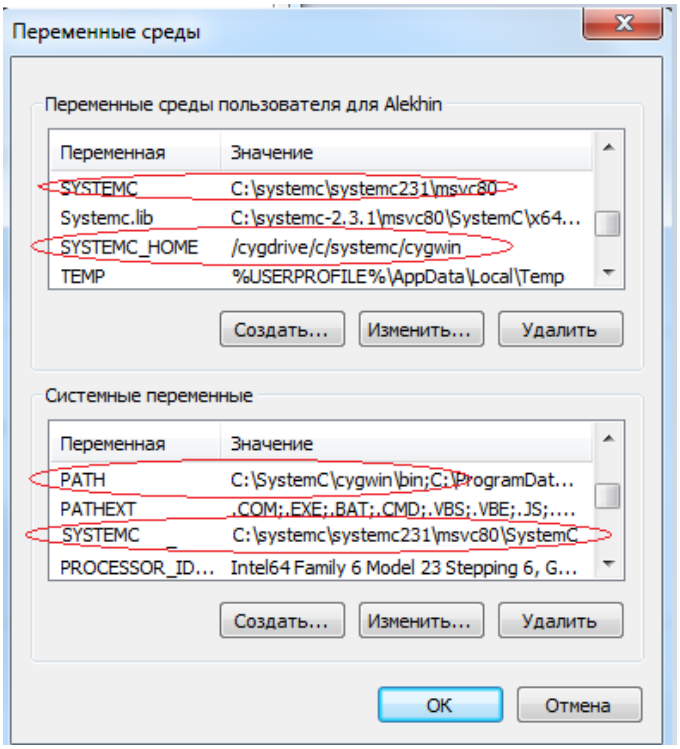


Рис. 2.16. Установка переменных сред и системных переменных для SystemC

2.1.7. Запуск Eclipse и настройки рабочего пространства

Далее с сайта http://www.eclipse.org загружаем архив программы Eclipse IDE for C++ Developers (Version: Neon).

Перед запуском Eclipse требуется установить программу Java, если эта программа не была установлена на Вашеи компьютере.

Устанавливаем Eclipse

1. Копируем распакованный архив в директорию C:\SystemC\eclipse и запускаем программу.

2. Выбираем директорию для рабочего пространства (рис. 2.17):

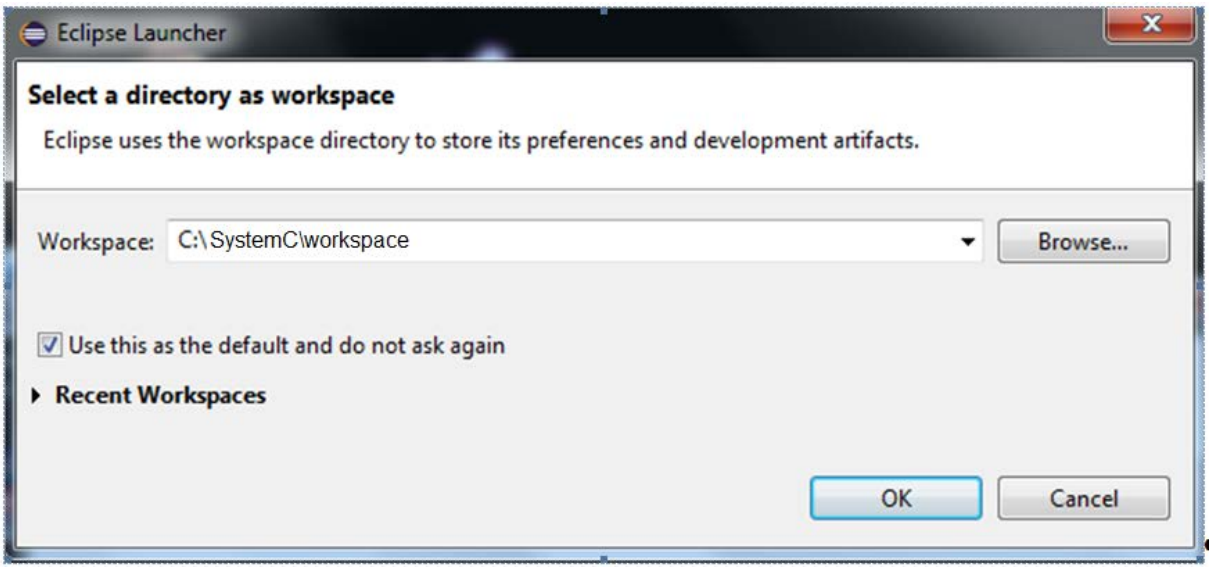


Рис. 2.17. Выбор директории для рабочего пространства

Выполняем настройки рабочего пространства:

3. В окне Windows выбираем Preferences. Далее выбираем General>Workspace и устанавливаем кодировку текста и New text file line delimiter (рис. 2.18).

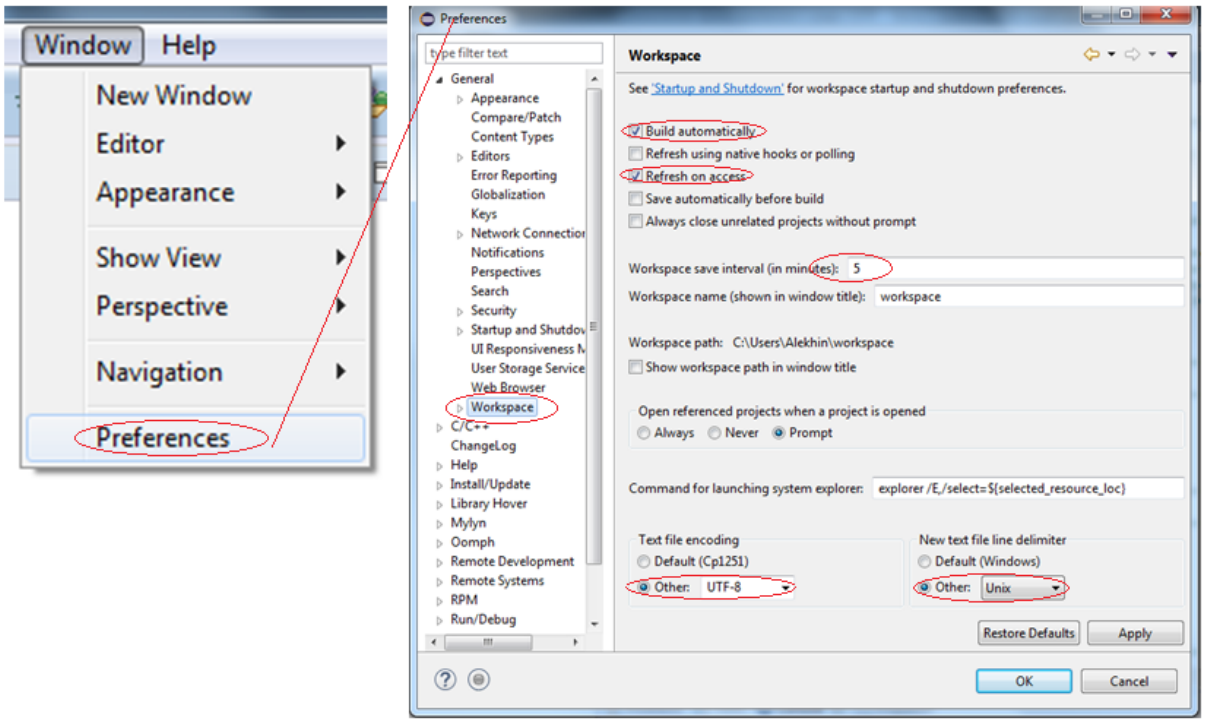


Рис. 2.18. Настройка рабочего пространства

4. На вкладке General выбираем C/C++ Project Wizard. Получаем доступные проекты и наборы инструментов, среди которых есть установленные нами Cygwin GCC. Выделяем Cygwin GCC и делаем его предпочтительным (рис. 2.19).

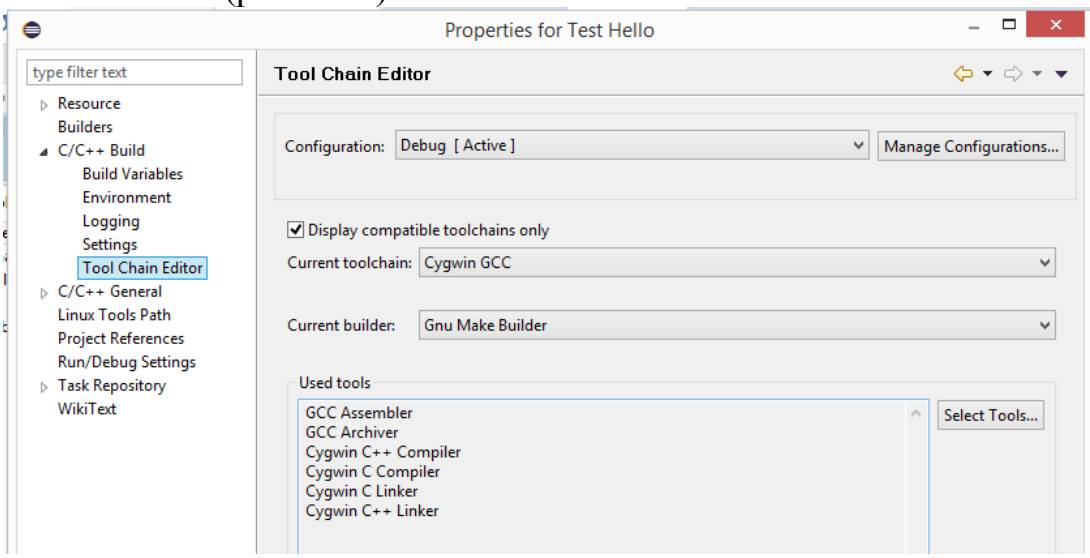


Рис. 2.19. Выбор компилятора

5. Проверяем установку переменных сред (рис. 2.20).

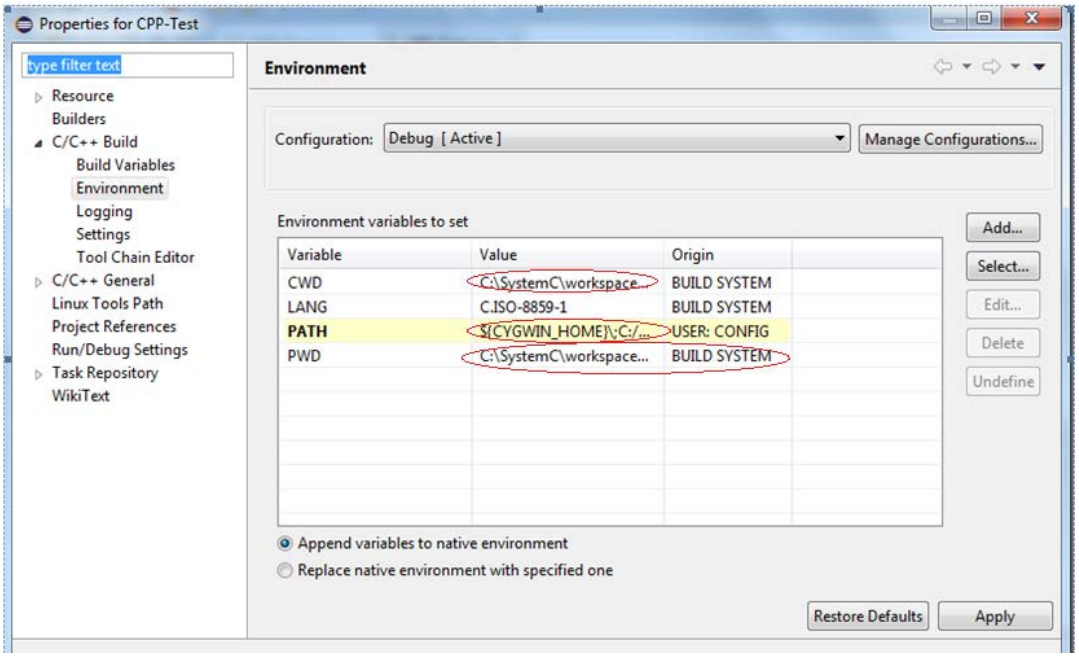


Рис. 2.20. Проверка переменных сред

6. На вкладке Text Editors устанавливаем параметры отображения текста (рис. 2.21).

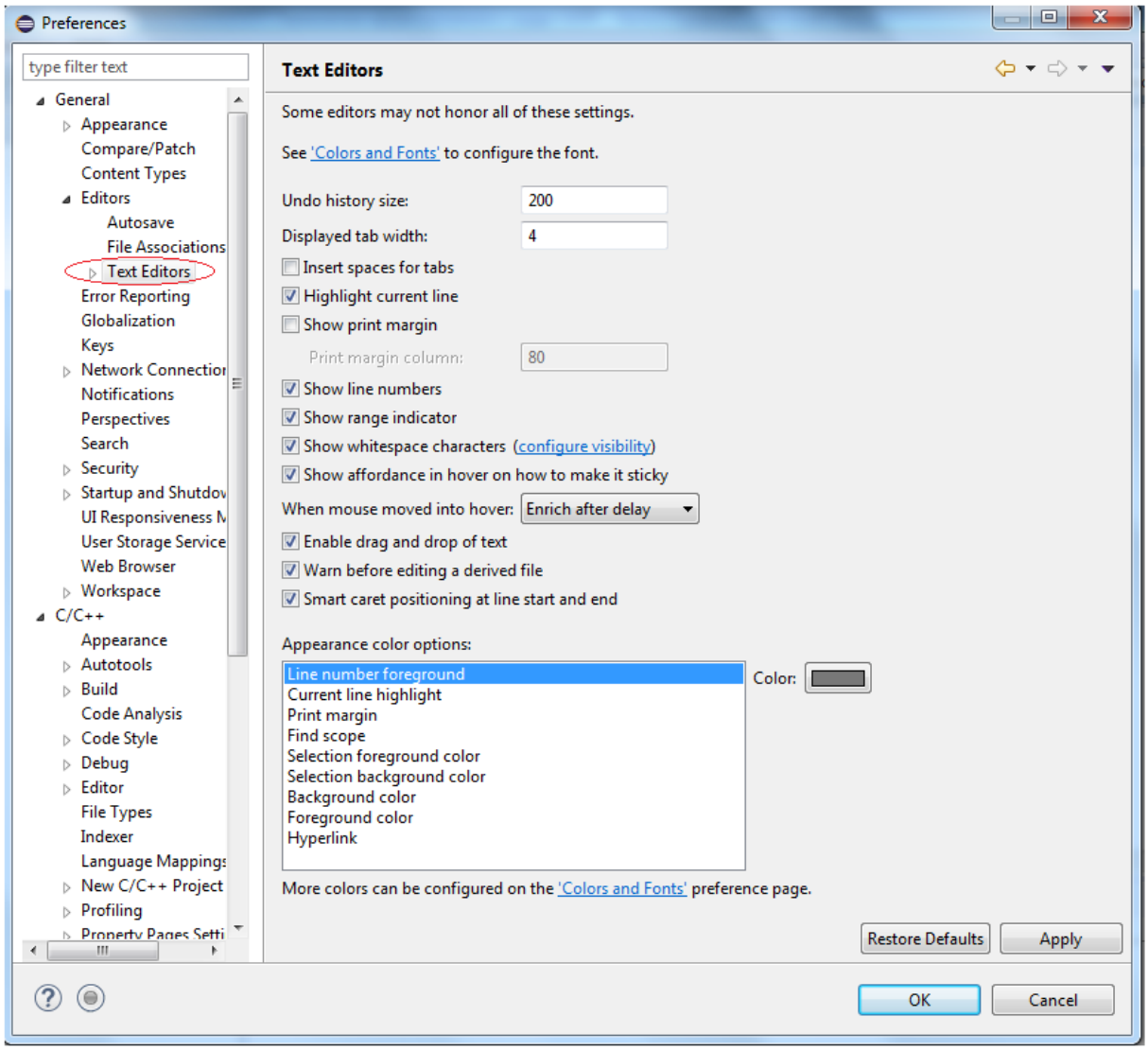


Рис. 2.21. Установка параметров текста

7. Далее выбираем C/C++ >Editor > Folding и устанавливаем разрешения на создание новых папок (рис. 2.22).

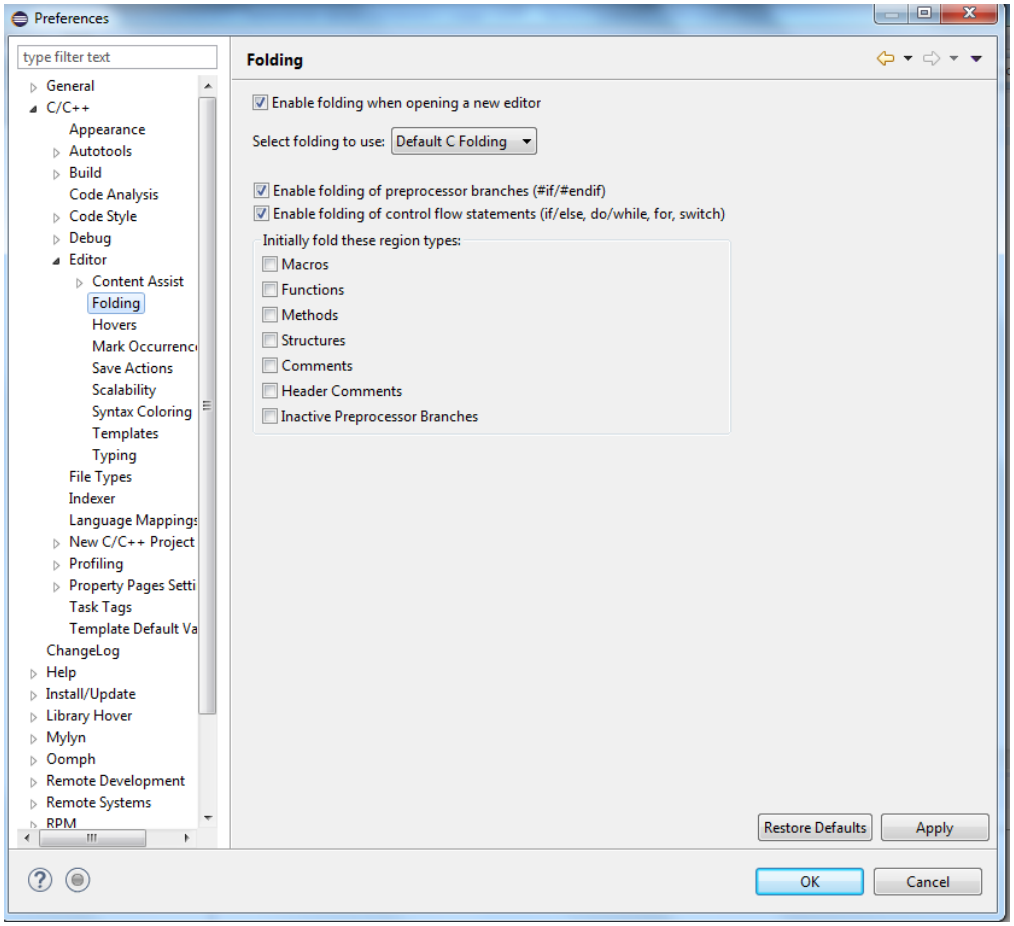


Рис. 2.22. Разрешение на создание новых папок

2.1.8. Создание нового проекта C++

1. В меню выбираем: File> New > C++Project (рис. 2.23).

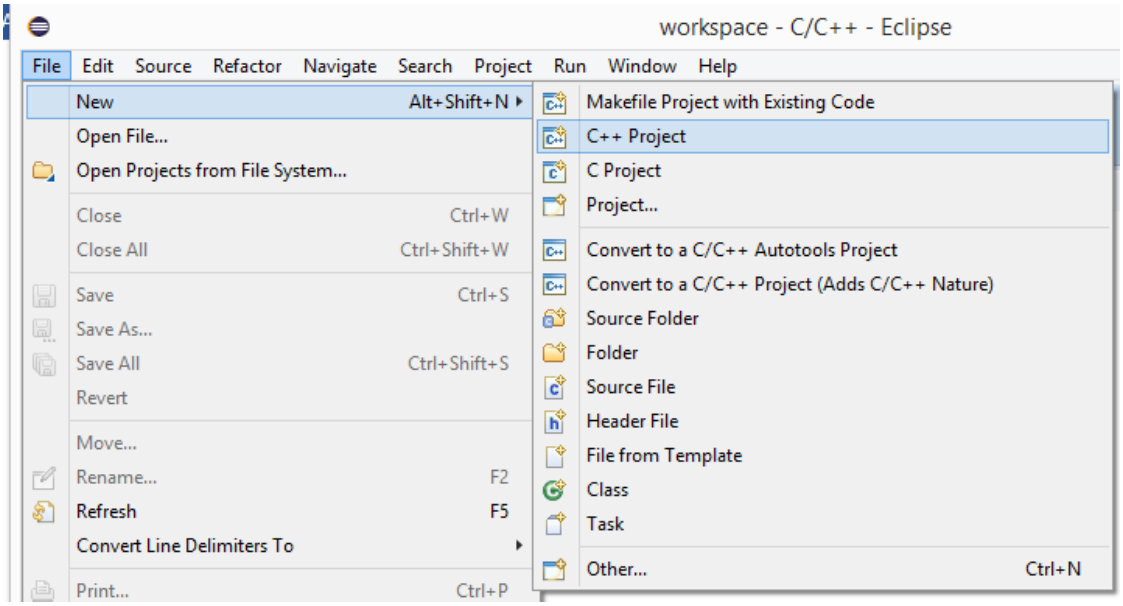


Рис. 2.23. Создание проекта C++

2. Задаем имя проекта, тип проекта «Hellow World C++Project», компилятор Cygwin GCC (рис. 2.24).

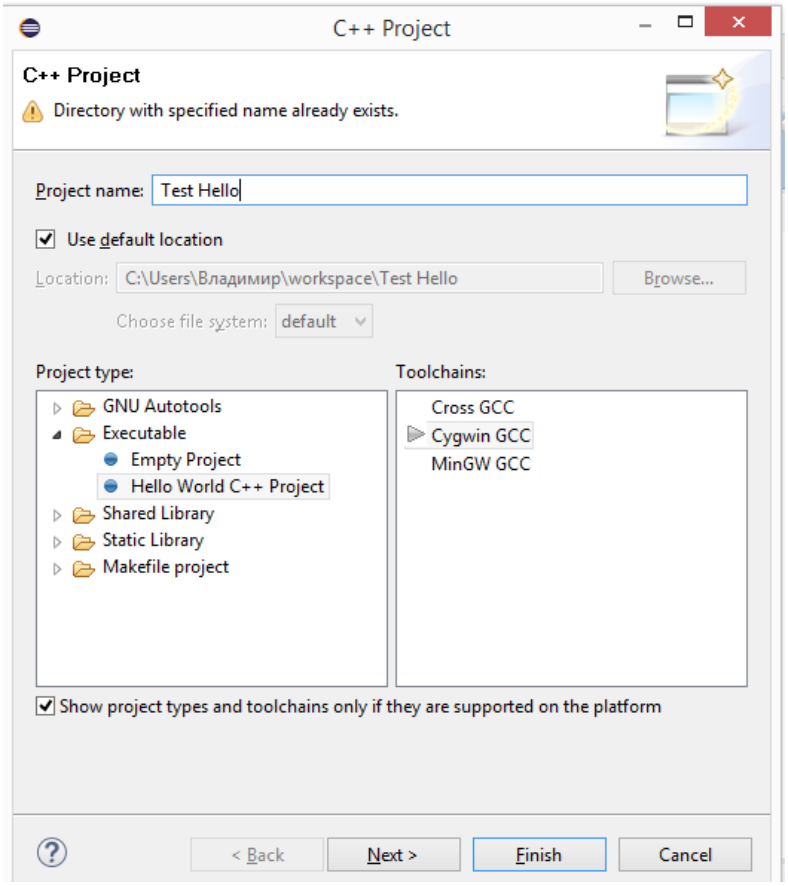


Рис. 2.24. Выбор типа проекта и компилятора

3. Выбираем папку src для хранения файлов (рис. 2.25).

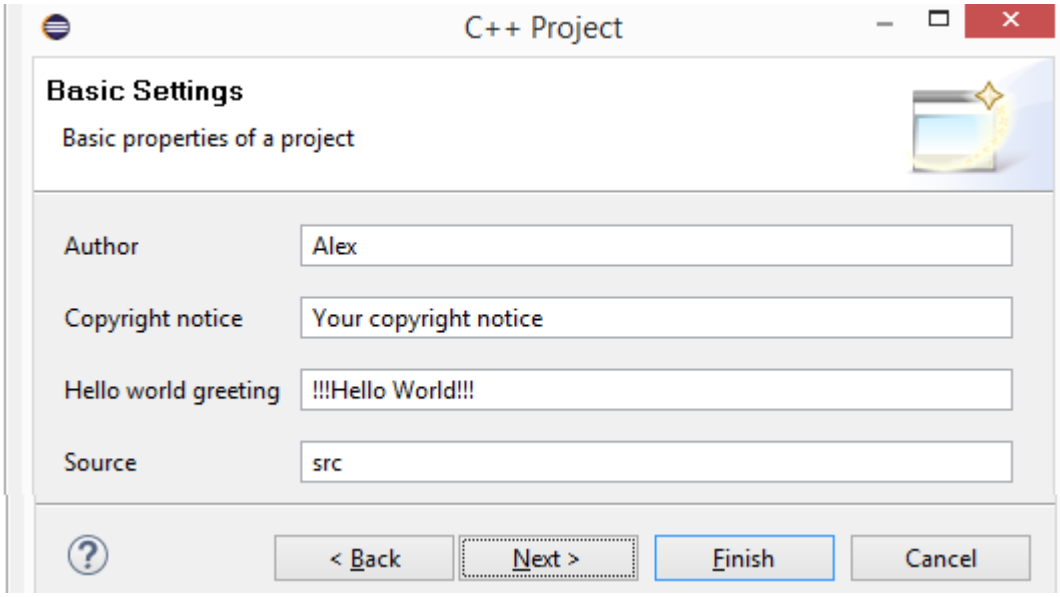


Рис. 2.25. Выбор папки src

4. Выбираем конфигурацию «Debug» и «Release» (рис. 2.26).

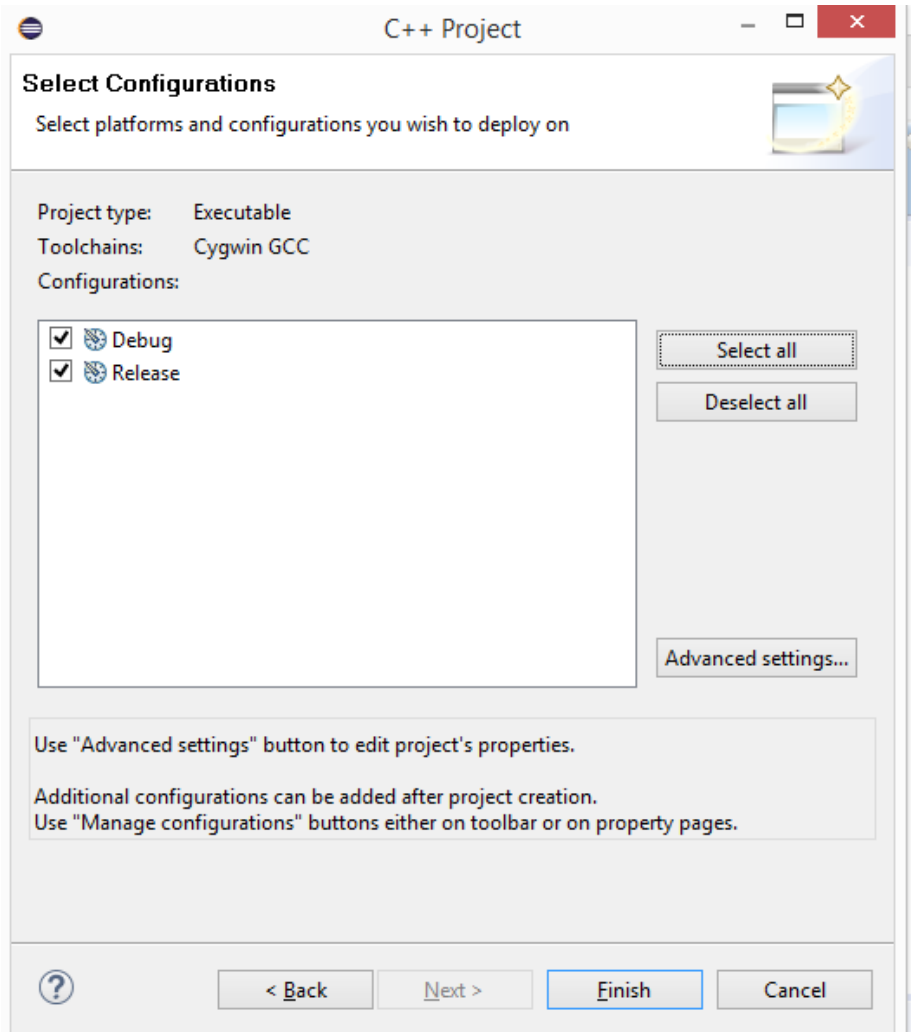


Рис. 2.26. Выбор конфигурации

5. Нажимаем Finish и получаем текст программы Test Hello.cpp. После компиляции (рис. 2.27) получаем исполняемый файл Binaries.

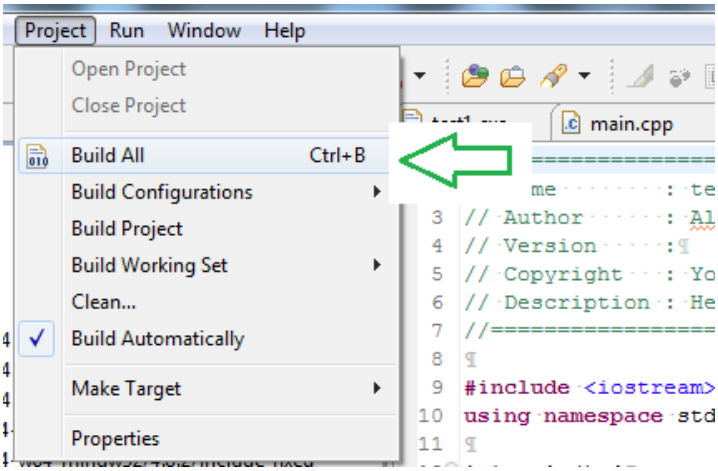


Рис. 2.27. Компиляция проекта

Проверяем ресурсы проекта (рис. 2.28). Для проекта C++ имеем 6 вложенных файлов, связанных с компилятором Cygwin.

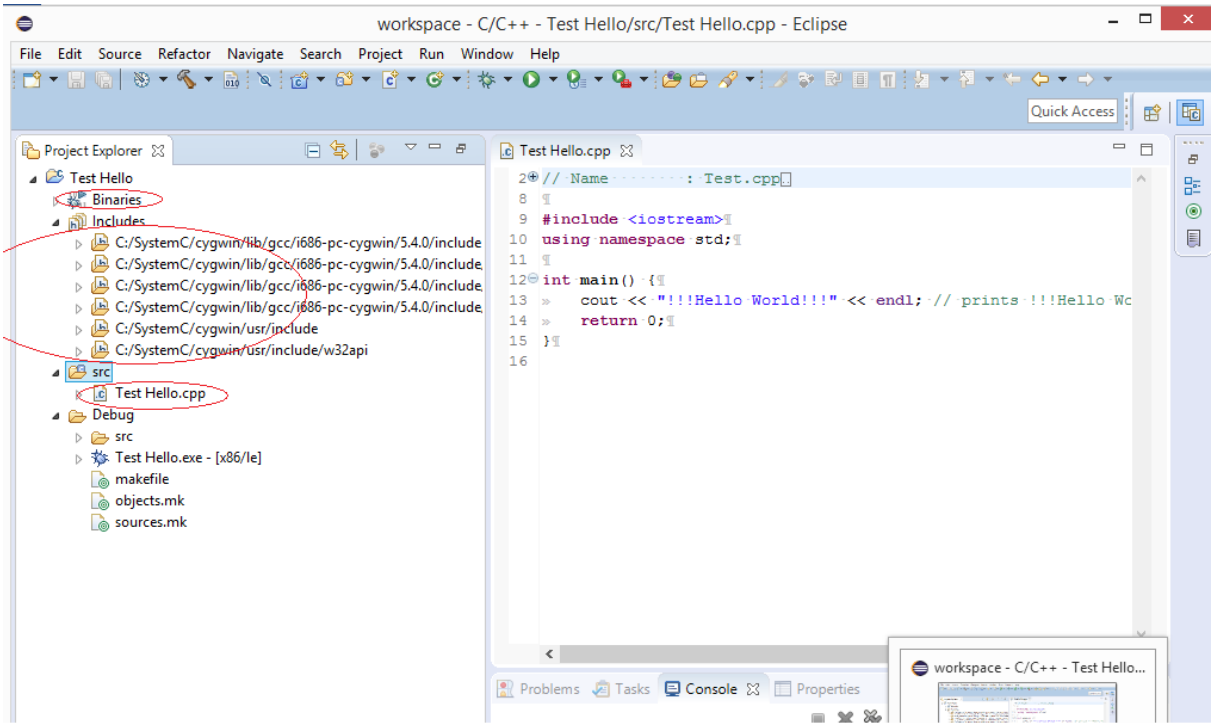


Рис. 2.28. Проверка ресурсов проекта C++.

6. Выполняем компиляцию и решение. Получаем в консоли Hello World! (рис. 2.29).

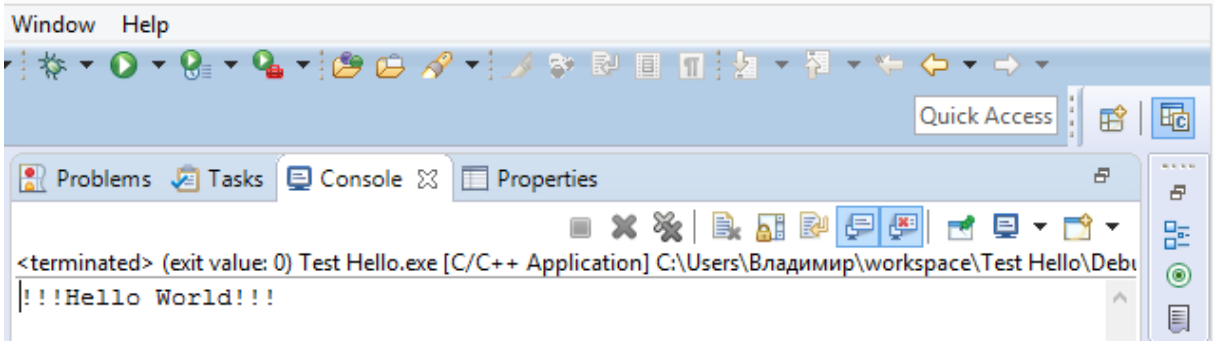


Рис. 2.29. Решение: !!!Hello World!!!

Теперь попробуем написать свою собственную программу и запустить ее. Для начала удалим весь шаблонный код из редактора и наберем следующий листинг:

```
#include <iostream>
using namespace std;

int main()
```

```
{

cout << "Hello, world!" << endl;
return 0;

}
```

Откомпилируем и запустим этот пример, в консоли если все настроено правильно и код был напечатан без ошибок должно отобразится сообщение приветствия `"Hello, world!"`. Теперь более подробно рассмотрим содержание этой программы.

```
#include <iostream>
```

Директива **#include** используется для подключения других файлов. Строка #include <iostream>, будет заменена содержимым файла «iostream.h», который находится в стандартной библиотеке языка и отвечает за ввод и вывод данных на экран.

```
using namespace std;
```

Содержимое третьей строки — `using namespace std;` указывает на то, что мы используем по умолчанию пространство имен с названием «std».

Пространство имен (*namespace*)— это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек.

Идентификатор, определённый в пространстве имён, ассоциируется с этим пространством. Один и тот же идентификатор может быть независимо определён в нескольких пространствах.

В языке C++ пространство имён определяется блоком инструкций:

```
namespace foo { int bar;

}
```

Внутри этого блока идентификаторы могут вызываться именно так, как они были объявлены. Но вне блока требуется указание имени пространства имён перед идентификатором. Например, вне `namespace foo`

идентификатор `bar` должен указываться как `foo::bar.` C++ содержит конструкции, делающие подобные требования необязательными. Так, при добавлении строки `using namespace foo;` в код, указывать префикс `foo::` больше не требуется.

В конце каждой команды ставится **точка с запятой.**

```
int main()
{
        …
}
```

Все то, что находится внутри фигурных скобок функции `int main() {}` будет автоматически выполняться после запуска программы. Данная функция является точкой входа в программу и должна присутствовать в любой программе, разрабатываемой на языке C++. В случае ее отсутствия компилятор будет выдавать ошибку.

```
cout << "Hello, world!" << endl;
```

Эта строка говорит программе выводить сообщение с текстом `"Hello, world!"` на экран.

Объект `cout` предназначен для вывода текста на экран командной строки. После него ставиться оператор вывода в поток - две угловые кавычки `(<<).` Далее идет текст, который должен выводиться. Он помещается в двойные кавычки, поскольку является строкой, далее снова следует оператор вывода и оператор конца строки – `endl`, который переводит строку на уровень ниже.

```
return 0;
```

Эта строка содержит оператор выхода и возвращения значения из функции, более подробно данная тема будет рассмотрена в теме «Функции».

Теперь попробуем скомпилировать следующую программу, ее листинг содержит уже известные нам операторы, за исключением оператора `cin`, который в отличи от `cout` наоборот выполняет ввод данных и помещает их в ячейку переменной указанную справа от оператора ввода из потока (>>). Листинг программы указан ниже:

```
#include <iostream.h>
int main()
{
```

```
    int day_now, month_now, year_now;

    cout << "Введите текущую дату и нажмите ENTER." <<
endl;
    cin >> day_now;
    cout << "Введите месяц и нажмите ENTER." << endl;
    cin >> month_now;
    cout << "Введите год и нажмите ENTER." << endl;
    cin >> year_now;
    cout << day_now <<”.”<< month_now <<”.”<<
year_now << endl << endl;

    return 0;

    }
```

Строка `int day_now, month_now, year_now;` - это операторы объявления переменных, данная тема будет рассмотрена далее в книге. Результат работы программы показан на рисунке 2.30.

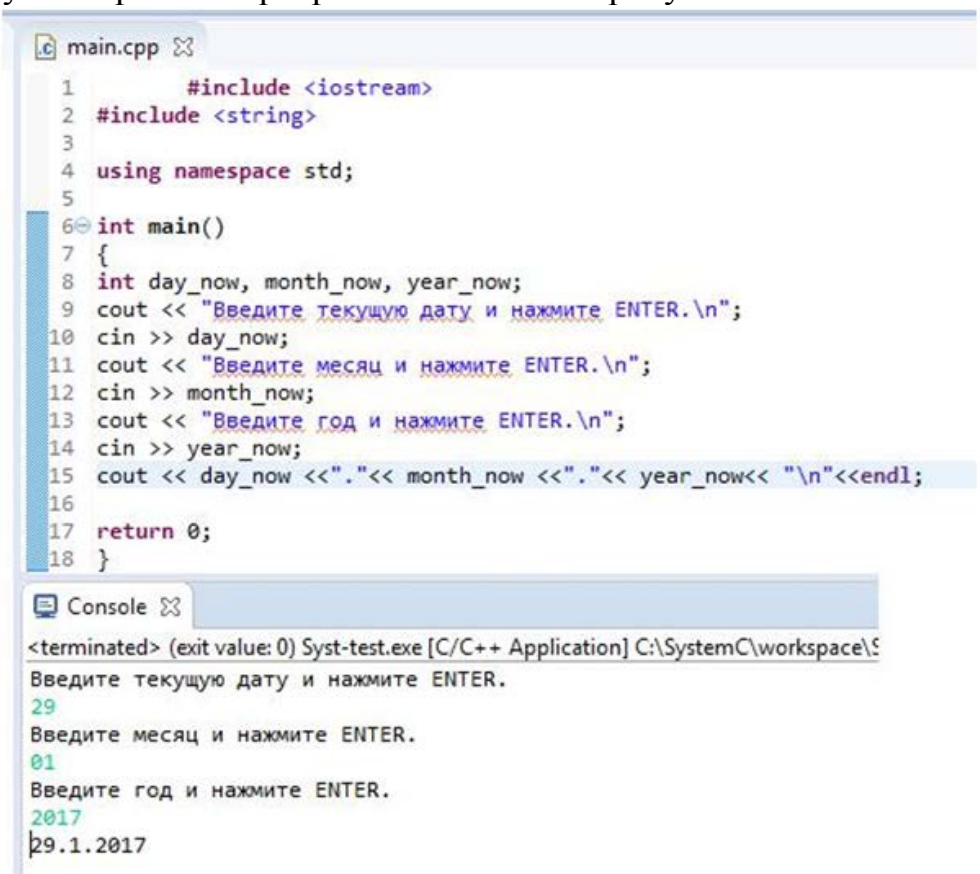


Рис. 2.30

2.1.9. Создание проекта «Hello-SystemC»

Проверим работу созданного проекта с файлом, написанном на языке SystemC. В этой программе комментарии записаны на русском языке.

1. Загружаем в папку src файл Hello-SystemC:

```
#include "systemc.h"
// Hello_world это имя модуля
SC_MODULE (hello_world) {
  SC_CTOR (hello_world) {
    // Ничего нет в конструкторе
  }
  void say_hello() {
    //Печать "Hello SystemC" в консоли
    cout << "Hello SystemC.\n";
  }
};

/* sc_main функция верхнего уровня, например, в
C++ main*/
 int sc_main(int argc, char* argv[]) {
  hello_world hello("HELLO");
  // Печать hello world
  hello.say_hello();
  return(0);
}
```

Подробный разбор этого примера будет сделан в следующих главах после изучения языка SystemC.

2. Подключаем библиотеки SystemC.

Выполняем Project>Properties>C/C++>Settings>Cygwin C++ Compiler>Includes вводим путь C:\SystemC\systemc231\include (рис. 2.31).

Нажимаем Apply-Ok.

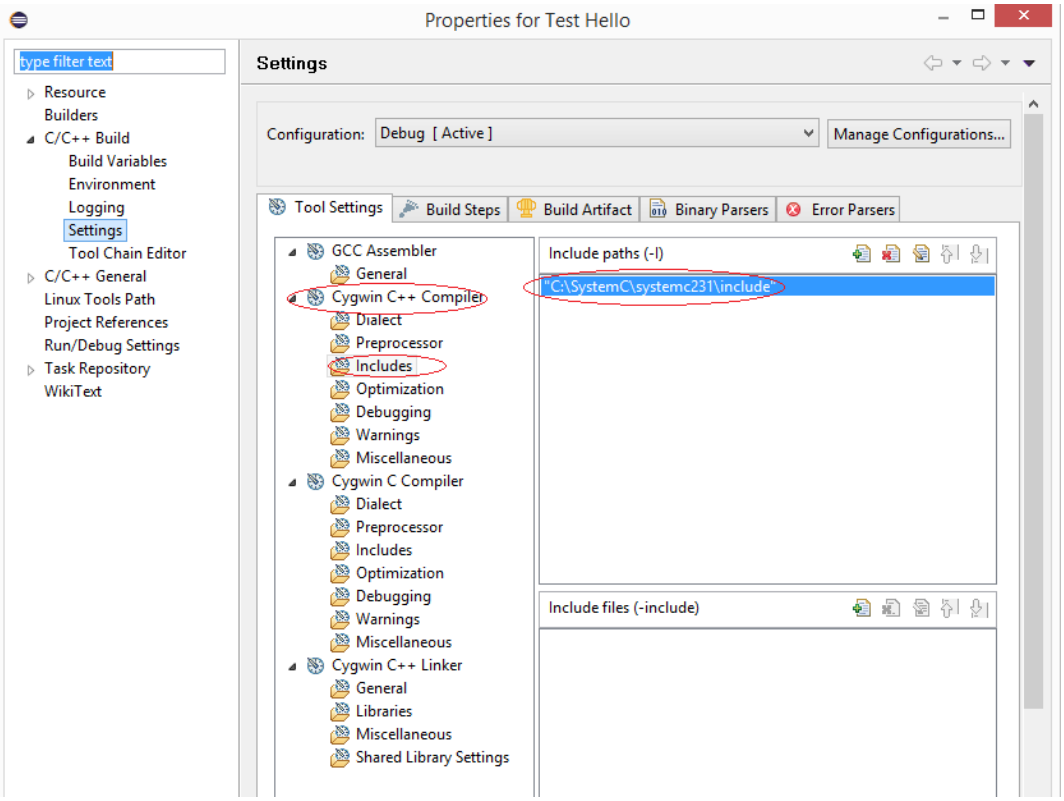


Рис. 2.31. Подключение библиотеки SystemC

3. Далее на вкладке Cygwin C++ >Linker> Libraires вводим SystemC и путь C:/SystemC/systemc231/Cygwin.lib (рис. 2.32).

Нажимаем Apply-Ok.

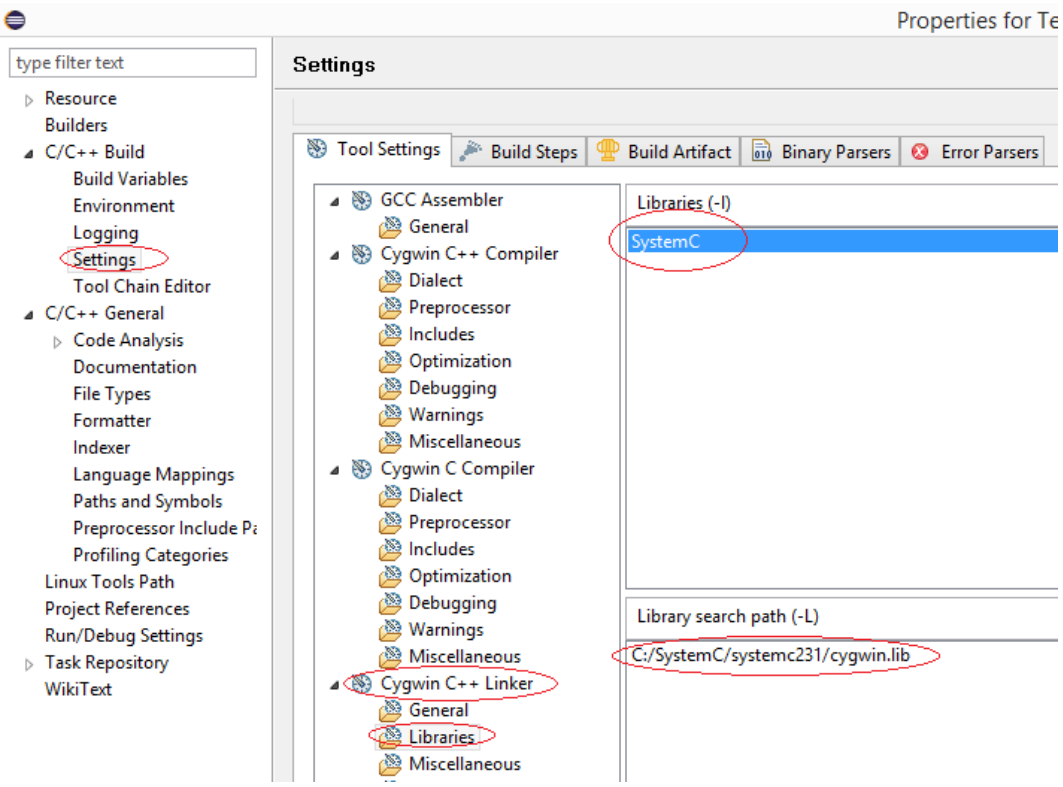


Рис. 2.32. Подключение библтотек к линкеру

4. Открываем файл SC-Hello из папки src, выполняем Save all , Build, Run. В консоли получаем Hello SystemC (рис. 2.33).

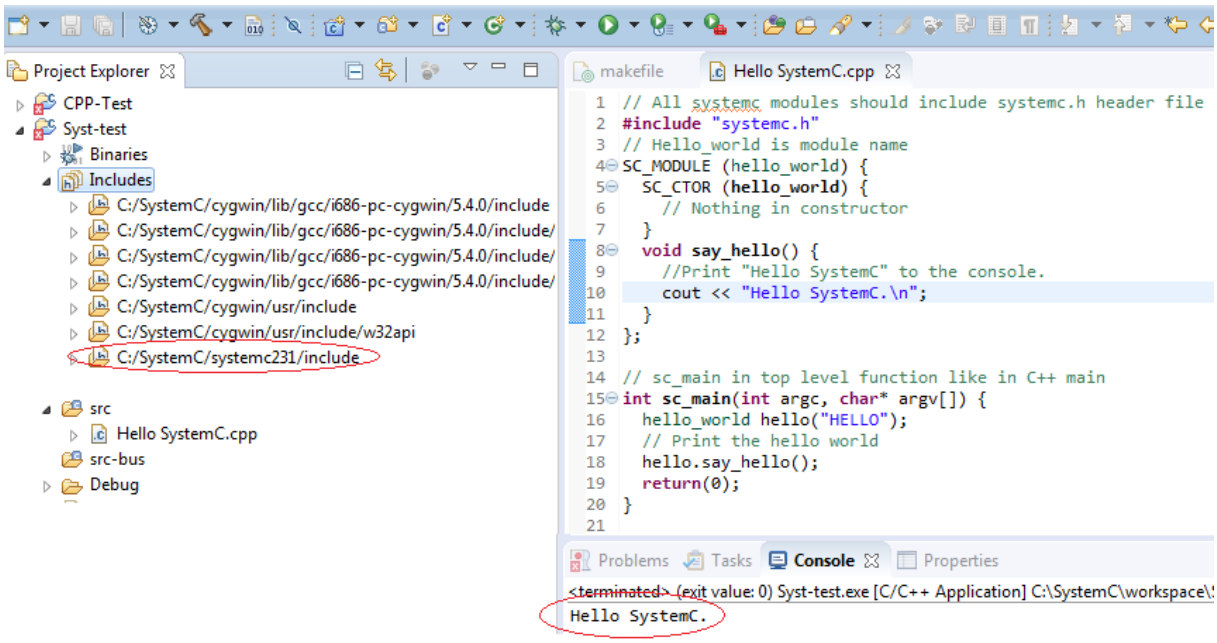


Рис. 2.33. Решение для Hello SystemC

Вложения Includes теперь дополнительно содержат библиотеки C:/SystemC/systemc231/include.

2.2. Как работать в Eclipse CDT

Eclipse, представляет собой интегрированную среду разработки (IDE) для Java и других языков программирования, таких как C, C++, PHP, и Ruby и т.д. Среда разработки, которая обеспечивается Eclipse, включает в себя средства разработки Eclipse Java (JDT) для Java, Eclipse CDT для C / C++, и Eclipse PDT для PHP, среди других.

Этот раздел научит вас, как использовать Eclipse при разработке любого программного проекта с использованием Eclipse IDE. Мы будем уделять особое внимание проекту Eclipse CDT для C / C++.

2.2.1. Главное меню, перспективы, workspace

На рис. 2.34 показано главное меню.

Меню **File** (Файл) позволяет открывать файлы для редактирования, закрывать редакторы, сохранить содержимое редактора и переименоввывать файлы. Среди прочего, он также позволяет выполнять импорт и экспорт содержимого рабочего пространства и завершать работы Eclipse.

В меню **Edit** (Правка) представлены предметы для копирования и вставки.

Меню **Source** (Источник) виден только тогда, когда редактор открыт. Она представляет собой ряд полезных пунктов меню, связанных с редактирование исходного кода.

Меню **Navigate** (Навигация) позволяет быстро находить необходимые ресурсы и затем перемещаться к ним.

Меню **Search** (Поиск) представляет элементы, которые позволяют искать рабочее пространство для файлов, содержащих определенные данные.

Project. Пункты меню, связанные с созданием проекта, можно найти в меню Project.

Run

Пункты меню в меню Run позволяет запускать программу в рабочем режиме или режиме отладки. В нем также представлены пункты меню, которые позволяют отлаживать код.

Window.

Меню Window позволяет открывать и закрывать виды и перспективы. Это также позволяет вызвать диалоговое окно настроек.

Help

Меню Help можно использовать, чтобы открыть окно справки, вид Eclipse Marketplace или установить новые плагины. Пункт меню о Eclipse, дает информацию о версии.

Чтобы переместить вид из одной папки вида к другому, просто нажмите на заголовке панели и перетащите его в область строки заголовка другой папки просмотра. Перемещение значка перетаскивания в нижней части окна позволяет создать вид папки, которая охватывает всю ширину окна. Перемещение значка перетаскивания левого или правого края окна позволяет создать папку просмотра, которая охватывает всю высоту окна.

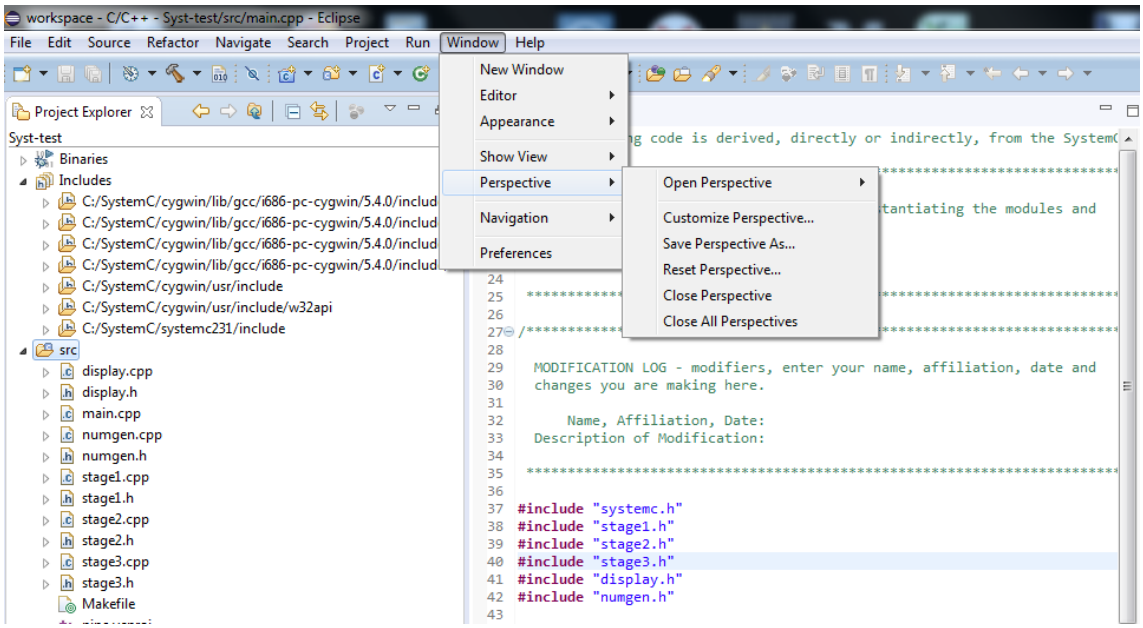


Рис. 2.34. Главное меню и рабочее поле редактора

На вкладке Window>Show View можно открыть разные окна (папки) для работы (рис. 2.35).

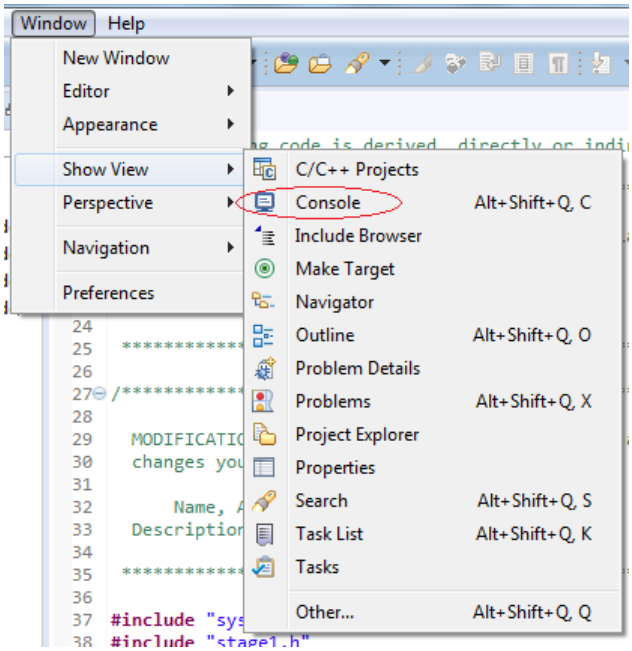


Рис. 2.35. Меню View

2.2.2. Перспективы в Eclipce

В Eclipse перспектива- это имя, данное к первоначальному сбору и расположению окон и областей редактора. Окно Eclipse может иметь несколько перспектив открытых в нем, но только одна перспектива активна в любой момент времени. Пользователь может переключаться между открытыми перспективами или открыть новую перспективу. Активная перспек-

тива управляет тем, что появляется в некоторых меню и панели инструментов. Открытие перспектив показано на рис. 2.36.

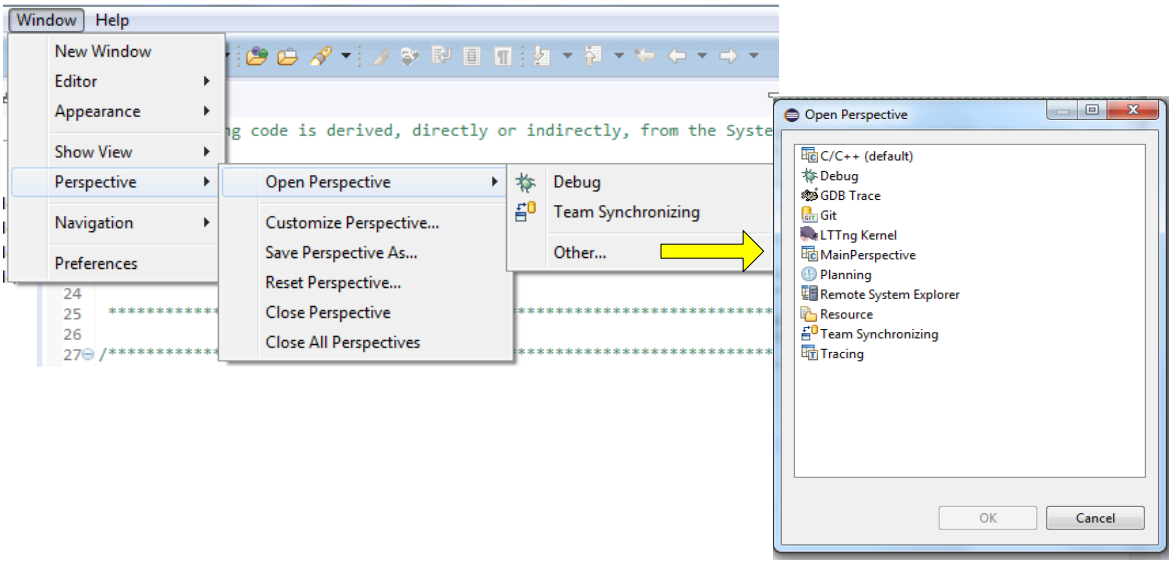


Рис. 2.36. Открытие перспектив

Workspace Eclipse (рабочее пространство) содержит ресурсы, такие как:

- Проекты
- Файлы
- Папки.

Рабочая область имеет иерархическую структуру. Проекты находятся на верхнем уровне иерархии, и внутри них вы можете иметь файлы и папки. Плагины используют API, предоставленную ресурсами плагина для управления ресурсами в рабочем пространстве.

Пользователи используют функциональные возможности, предоставляемые видом, редактором и мастером для создания и управления ресурсами в рабочем пространстве. Одним из многих представлений, которые показывают содержание рабочей области, является представление Project Explorer.

Мастер файла (File > New > File) можно использовать для создания нового файла.

Мастер папок (File → New → Folder) можно использовать для создания новых папок.

Хотя Eclipse, написана на Java, Eclipse, был разработан таким образом, что он может быть использован для поддержки нескольких языков, а не только Java. Программы C и C ++ могут быть разработаны на установленном плагине CDT.

Перед тем, как начать разрабатывать проекты C/C++ , необходимо сначала установить соответствующий компилятор C/C++ в вашей системе. Есть несколько компиляторов с открытым исходным кодом, доступных для использования с Windows, большинство основанно на проекте Unix-GCC. Мы используем компилятор Cygwin.

Для запуска C/C++ программ, необходимо установить Eclipse CDT (C/C++ Development Tools) плагин. Этот плагин можно загрузить со страницы Eclipse CDT (http://www.eclipse.org/cdt/).

Для того, чтобы гарантировать, что CDT был установлен правильно, выберите Help>About Eclipse SDK. Затем нажмите на кнопку Eclipse Installation Details. Вы должны увидеть CDT в верхней части списка (рис. 2.37).

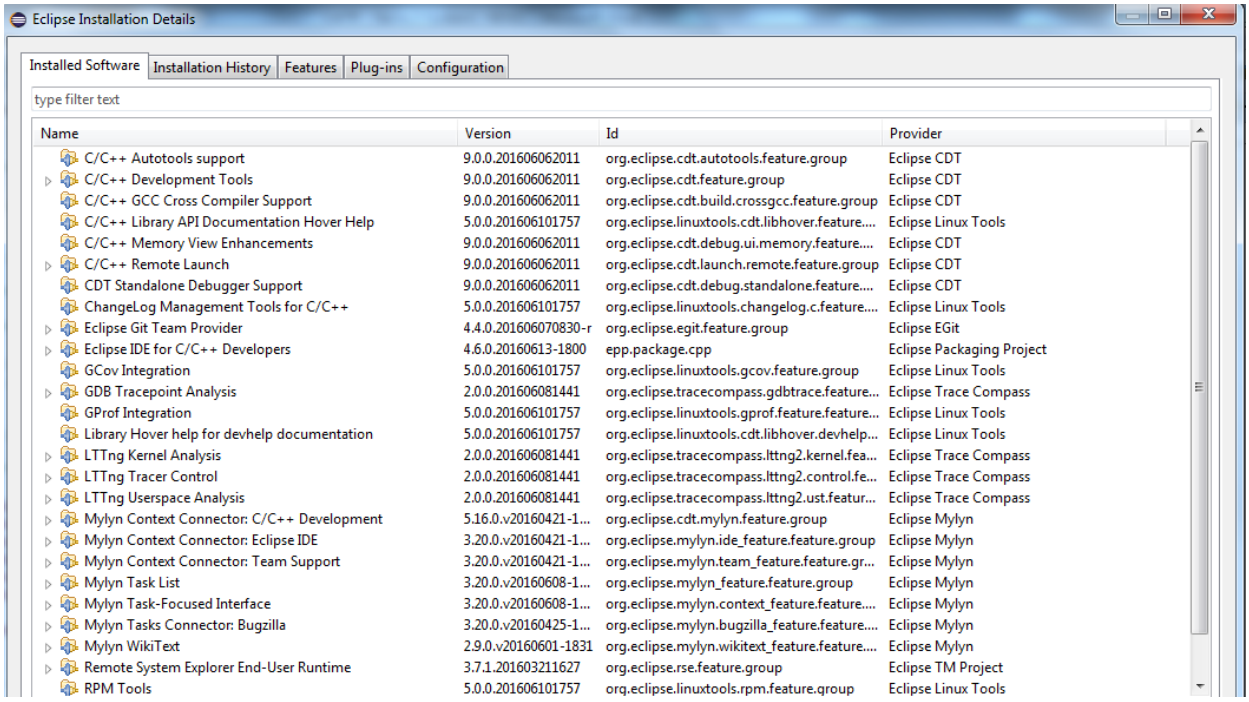


Рис. 2.37. Проверка установки Eclipse CDT

2.3. Программа GTKWave

При моделировании в SystemC часто требуется выводить в графическом виде временные диаграммы и в численном виде сохраненные массивы результатов моделирования. Для этого используют программу GTKWave.

GTKWave является инструментом анализа, используемым для выполнения отладки имитационных моделей на Verilog или VHDL. За исключением интерактивного просмотра файлов с расширением .VCD, GTKWave не предназначен для запуска в интерактивном режиме с моделированием, но вместо этого позволяет просматривать файлы дампов.

Поддерживаются различные DUMPFILE форматы:

VCD: Value Change Damp. Это стандартный промышленный формат файла генерируется большинством Verilog тренажеров и указывается в IEEE-1364. Он является самым медленным из форматов для наблюдения процесса и требует большой памяти, однако формат повсеместно используется и почти все инструменты поддерживают его.

Перечислим без комментариев прочие поддерживаемые форматы: LXT, LXT2, VZT, GHW, AET2, IDX, FST, VPD, WLF, FSDB.

Вспомогательные приложения «Конвертер» поставляются вместе с GTKWave чтобы преобразовать VCD файлы в форматы LXT, LXT2, VZT или FST файлов. Преобразование из LXT2, VZT и FST обратно в VCD также возможно.

GTKWave был разработан для выполнения отладки задач больших систем на чипе и был использован в этом качестве в качестве автономной замены для отладочных инструментов третьих фирм. Это 64-битная программа готова к самым крупным из проектов, учитывая, что выполняется на рабочей станции с достаточным количеством физической памяти. Файл - форматы LXT2 и VZT были специально разработаны, чтобы справиться с большими, реальные проектами, а AET2 (доступно только для пользователей инструмента IBM EDA) и FST были разработаны, чтобы эффективно обрабатывать очень большие проекты.

Для Verilog, GTKWave позволяет пользователям отлаживать результаты моделирования как на уровене сети, обеспечивая вид с высоты птичьего полета множества значений сигнала в течение различных периодов времени, а также на уровне RTL через аннотации значений сигнала обратно в библиотеку RTL для данного временного шага. Браузер RTL освобождает пользователей от необходимости иметь дело с фактическим местом, где данный модуль находится в RTL, как вид предоставляемой RTL браузером по умолчанию на уровне модуля. Это обеспечивает быстрый доступ к модулям в RTL, так как навигация была уменьшена просто до движения вверх и вниз в виде древовидной структуры, которая представляет фактический дизайн.

GTKWave как совокупность двоичных файлов состоит из двух взаимосвязанных средств: gtkwave приложение для просмотра и rtlbrowse.

Кроме того, коллекция Help-приложения используются для облегчения таких задач, как преобразование файлов и моделирования сбора данных. Они предназначены для совместной работы в совмещенной системе, хотя их модульная конструкция позволяет каждому работать независимо друг от друга, если это будет необходимо.

Gtkwave является анализатором сигнала и является основным инструментом, используемым для визуализации. Он предоставляет метод для просмотра результатов моделирования для аналоговых и цифровых

данные, пригодным для различных поисковых операций и временных манипуляций, можно сохранить частичные результаты (то есть "сигналы, представляющие интерес"), извлеченные из полного дампа моделирования, и, наконец, может генерировать PostScript и FrameMaker выход для твердой копии.

Rtlbrowse используется для просмотра и навигации по RTL исходному коду, который анализируется и обрабатывается в «стеблях» файлов по вспомогательным «паразитным» приложениям. Это пригодно для просмотра RTL как на уровне файлов и модуля, так и когда вызывается gtkwave, для аннотации исходного кода.

Вспомогательные приложения выполняют различные специализированные задачи, такие как преобразование файлов, RTL синтаксического анализа, а также другие операции манипулирования данными рассматриваемые вне сфера того, что необходимо выполнить инструменту визуализации.

2.3.1. Главное окно GTKWave

Главное окно GTKWave (рис. 2.38) инструмент визуализации состоит из секции строки меню, окна состояния, нескольких групп кнопок, раздела состояния времени, и секций сигнал и значение волны. Новым в GTKWave 3.0 является включение вложенного «дерева поиска сигнала» (SST), открывающегося слева от секции сигнала. Изображение, как правило, появляется, как показано ниже, когда отключается встроенный SST.

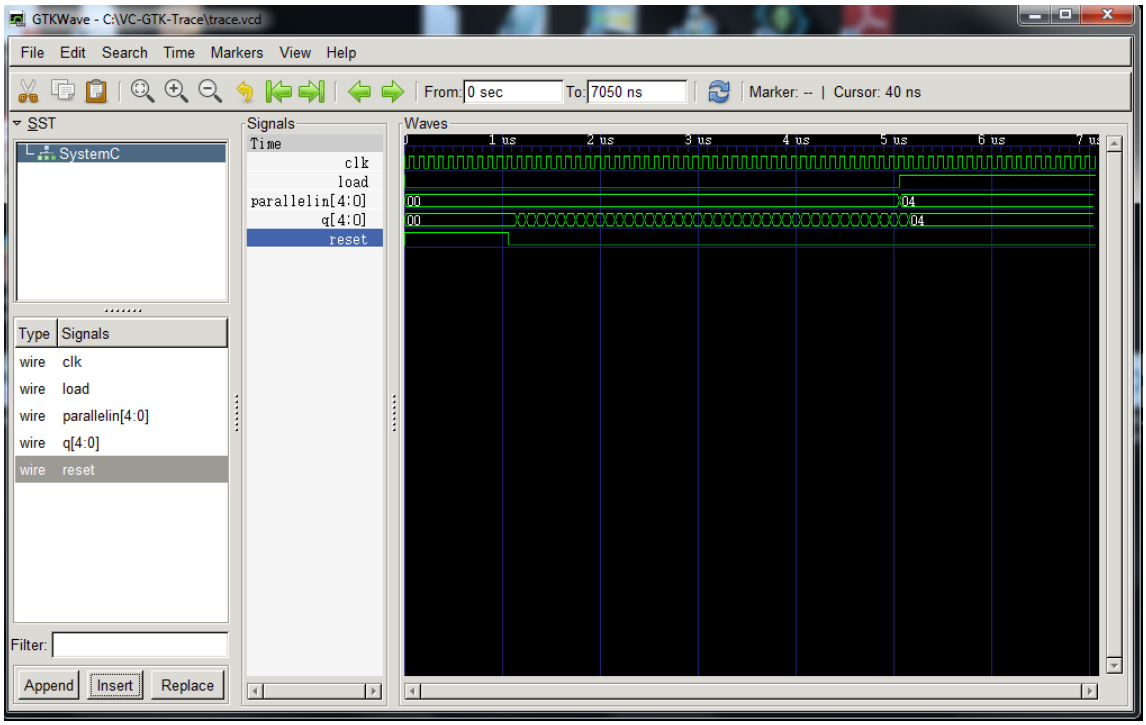


Рис. 2.38. Главное окно GTKWave

Кнопка Insert вставляет выделенный сигнал в окно Wave. Кнопка Append добавляет выделенный сигнал в конец окна отображения. Кнопка Replace перемещает выделенный сигнал над сигналом, выделенным выше.

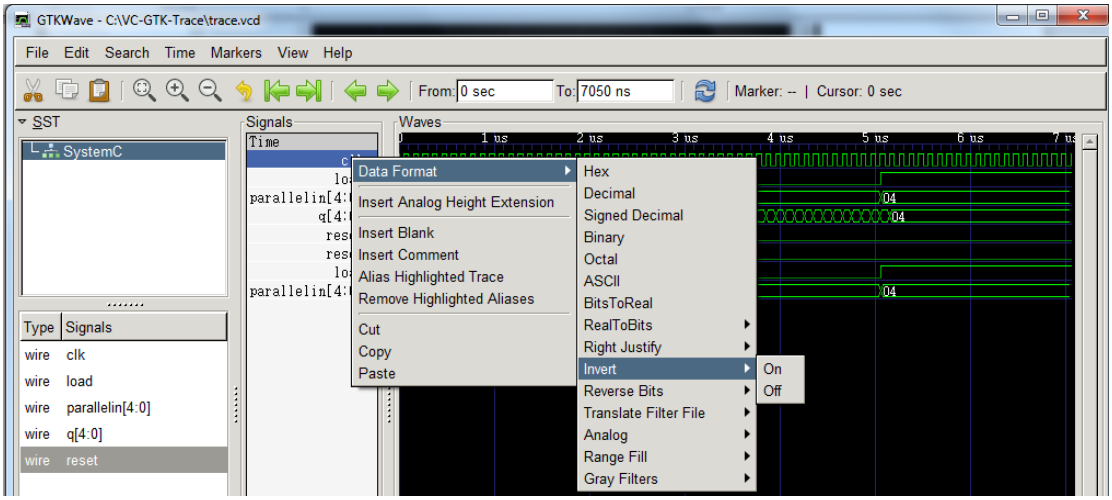


Рис. 2.39. Включение выбора формата чисел

Сигналы можно представлять в разных форматах данных (рис. 2.39), вставлять пустые строки, инвертировать, вставлять комментарии и пр. (рис. 2.40)

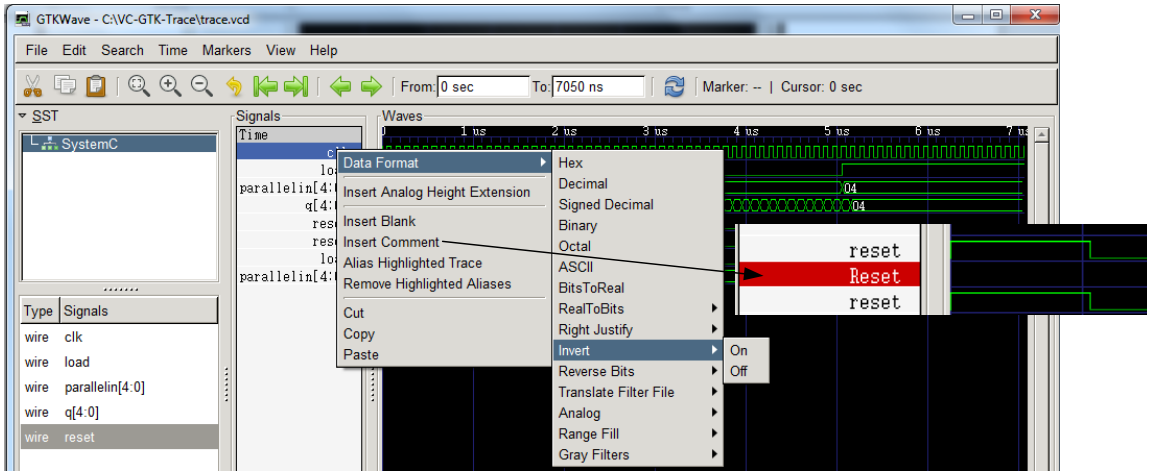


Рис. 2.40. Упраление форматом

С версии GTK больше или равной 2,4, встроенный SST позволяет перетаскивать сигналы с панели "Сигналы" внутри SST в панель "Сигналы" снаружи SST, что является удобным способом импортировать сигналы в окно наблюдения.

2.3.2. Маркеры и масштабы

С помощью маркеров можно измерять уровень сигналов в различных форматах данных. Над экраном отображается положение маркера и курсора (рис. 2.41).

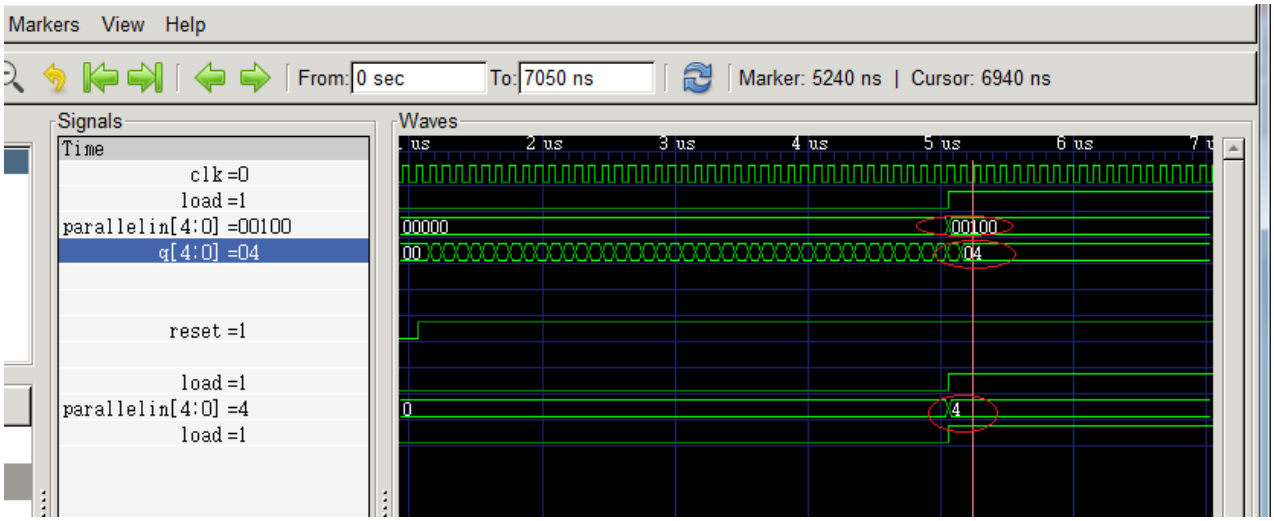


Рис. 2.41. Установка маркеров

В окне Signals отображаются все сигналы и их численные значения в позиции маркера. Расширение окна позволяет отображать более длинные числа.

Размеры окна, положение сигналов и маркера могут быть сохранены между сессиями.

Текущее положение маркера можно зафиксировать, выбрав Drop Named Marker, а затем, выбрав Show-Change Marker Data, показать в таблице до 26 именованных маркеров (рис. 2.42).

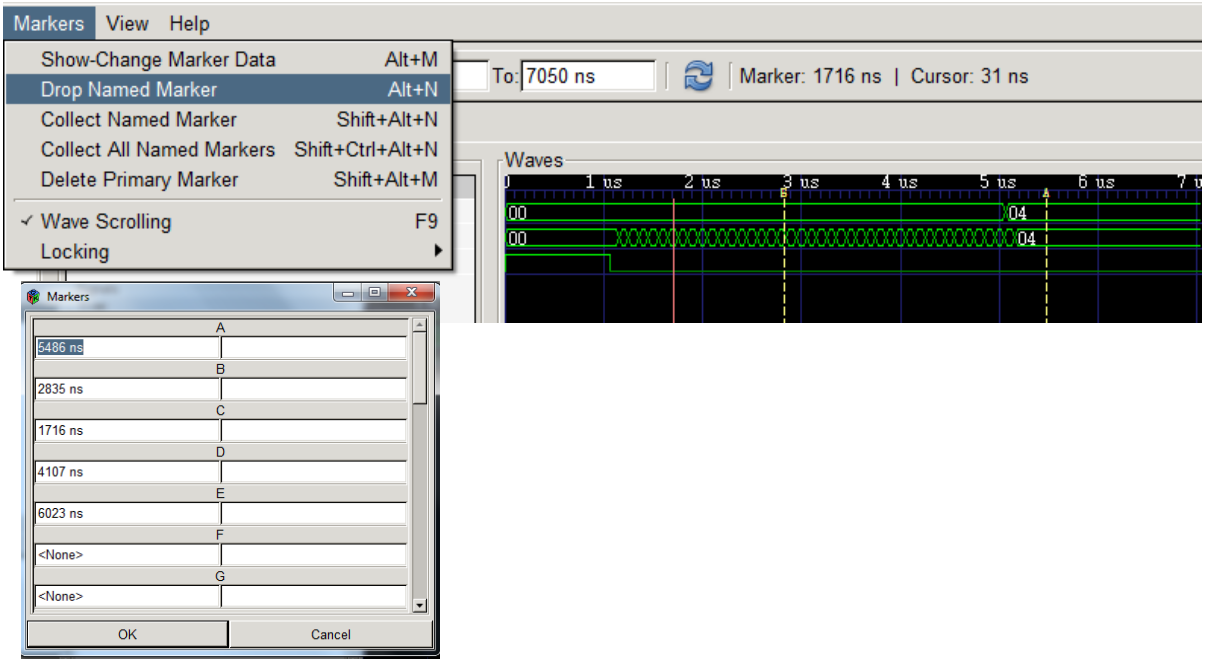


Рис. 2.42. Чтение результатов в маркеров

Командами Collect Named Markers можно убрать маркеры из окна наблюдения.

2.3.3. Главное меню

На рис. 2.43 показаны кнопки управления масшабом.

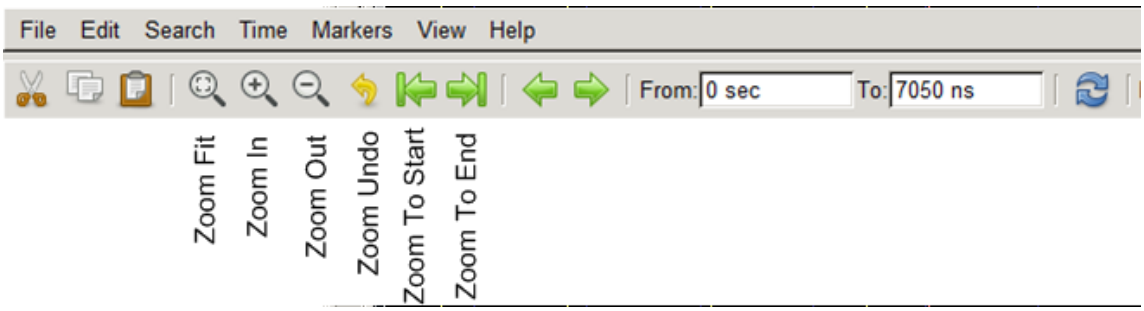


Рис. 2.43. Управление масштабом

На рис. 2.44 показаны открытые панели главного меню.

В меню File команда **Read Save File** будет открывать запрошенный файл по имени сохраненного в GTKWave файла. Содержимое файла сохранения будет определять, как трассы так и векторы, а также их формат (двоичный, десятичный, шестнадцатеричный, обратный ход и т.д.), которые должны быть отражены на дисплее. Обратите внимание, что маркер, позиционные данные и коэффициент масштабирования, присутствующие в сохраненном файле, заменят все текущие настройки.

Write Save File файл будет вызывать Write Save File As, если имя сохраненного имя файла не было указано ранее. В противном случае он будет писать сохраненые данные файлов без запроса.

Write Save File As будет открыть сохраненный файл, имя которого запросит GTKWave. Содержимое сгенерированного сохраненного файла включает трассы, а также их формат (двоичный, десятичный, шестнадцатеричный, обратный ход и т.д.), которые в настоящее время являются частью дисплея. Позиционные данные маркера и коэффициент масштабирования также являются частью сохранения файла.

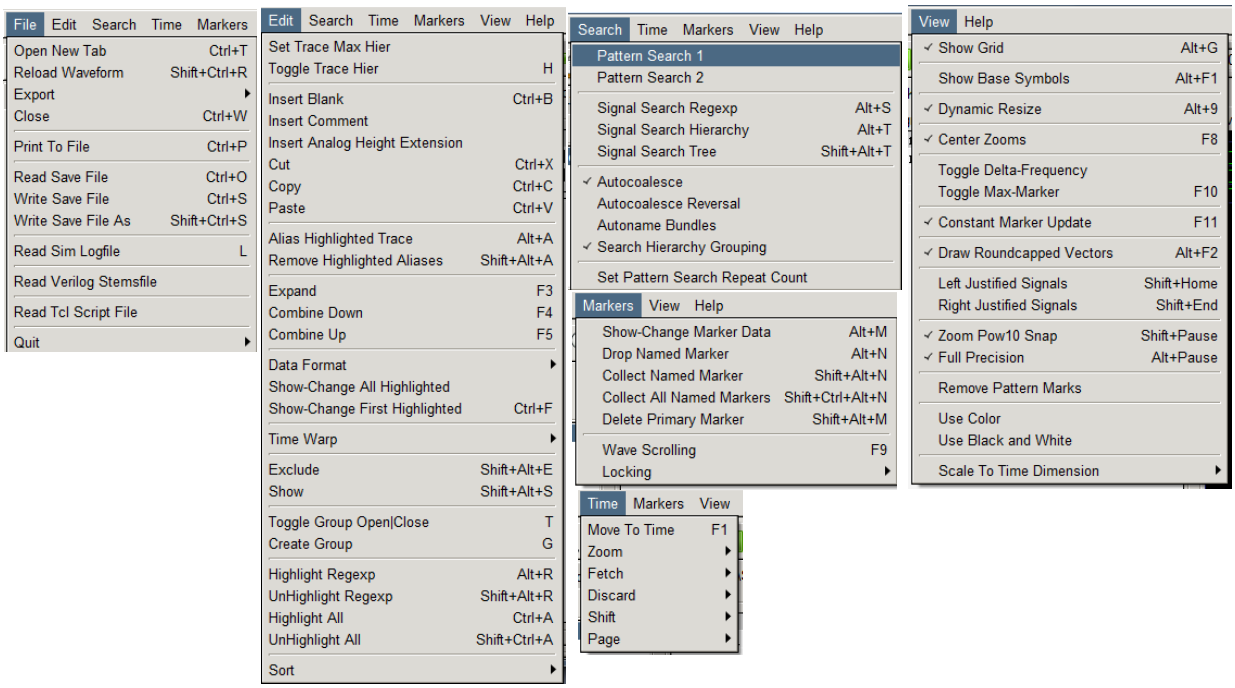


Рис. 2.44. Панели главного меню

Режим **TwinWave** является последним достижением GTKWave, что позволяет открыть две сессии в одно время в одном окне. Горизонтальная прокрутка, коэффициент масштабирования, первичный маркер, и вторичный маркер синхронизированы между двумя сессиями.

Для перехода в этот режим просто вызовите twinwave с аргументами для каждой сессии gtkwave, перечисленными полностью, разделяя их со знаком плюс:

twinwave a.vcd a.sav + b.vcd b.sav

Более подробную информацию Вы найдете в GTKWave 3.3 Wave Analyzer User's Guide.

2.3.4. Установка и использование GTKWave

1. Для установки GTKWave с сайта GTK надо загрузить GTKWave v.3.3.8 для WIN32 (gtkwave.exe.gz) и выполнить распаковку архива, дважды использую 7-ZIP в папку C:/gtk (рис. 2.45).

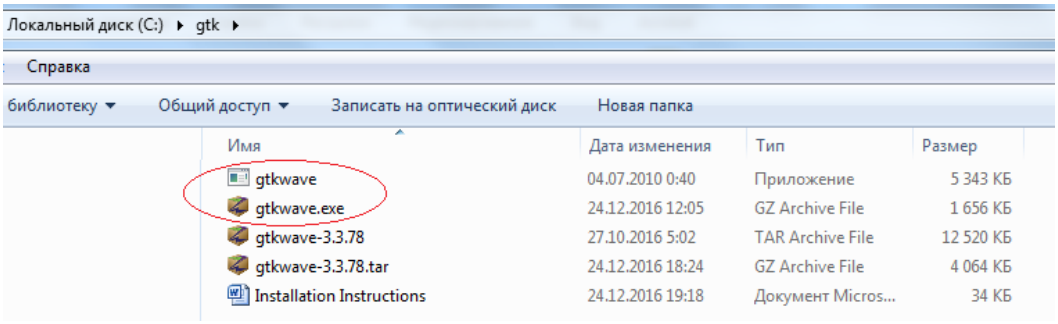


Рис. 2.45. Загруженные файлы GTKWave

2. Загружаем архив всех библиотек (all_libs.tar.gz) в папку C:/gtklib и выполняем распаковку архива в этой папке (рис. 2.46).

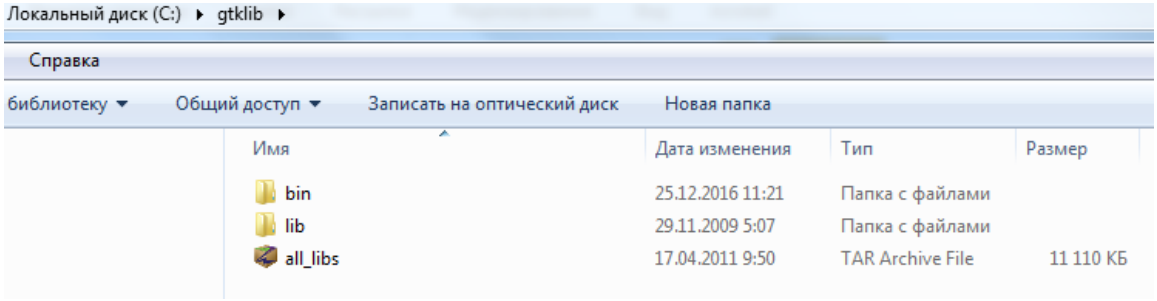


Рис. 2.46. Содержание папки gtklib

3. Копируем файл gtkwave.exe из папки C:/gtk в папку C:/gtklib/bin (рис. 2.47).

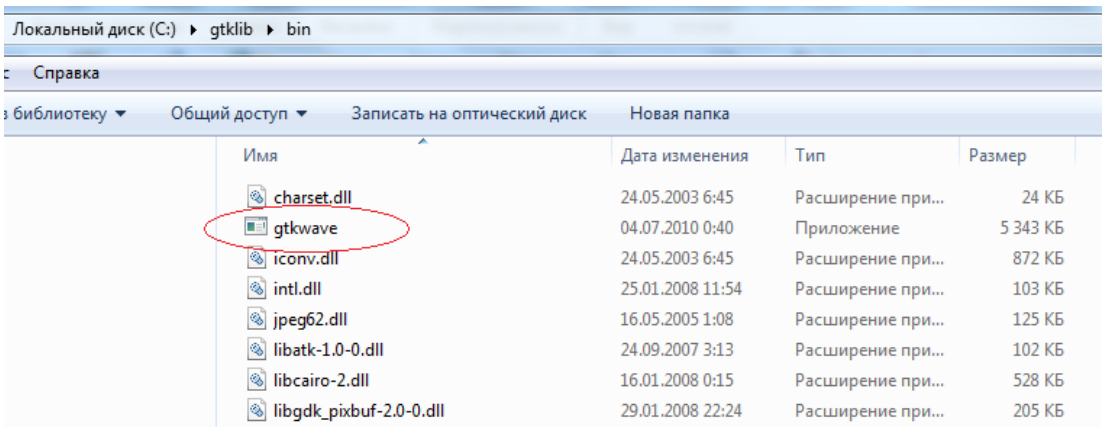


Рис. 2.47. Содержание C:/gtklib/bin

4. Добавляем путь в переменные среды PATH=C:\gtklib\bin (рис. 2.48).

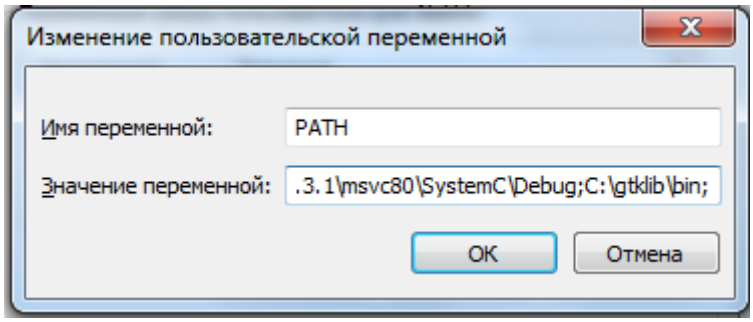


Рис. 2.48. Изменение пользовательской переменной

5. Отправляем файл запуска на рабочий стол и запускаем программу (рис. 2.49).

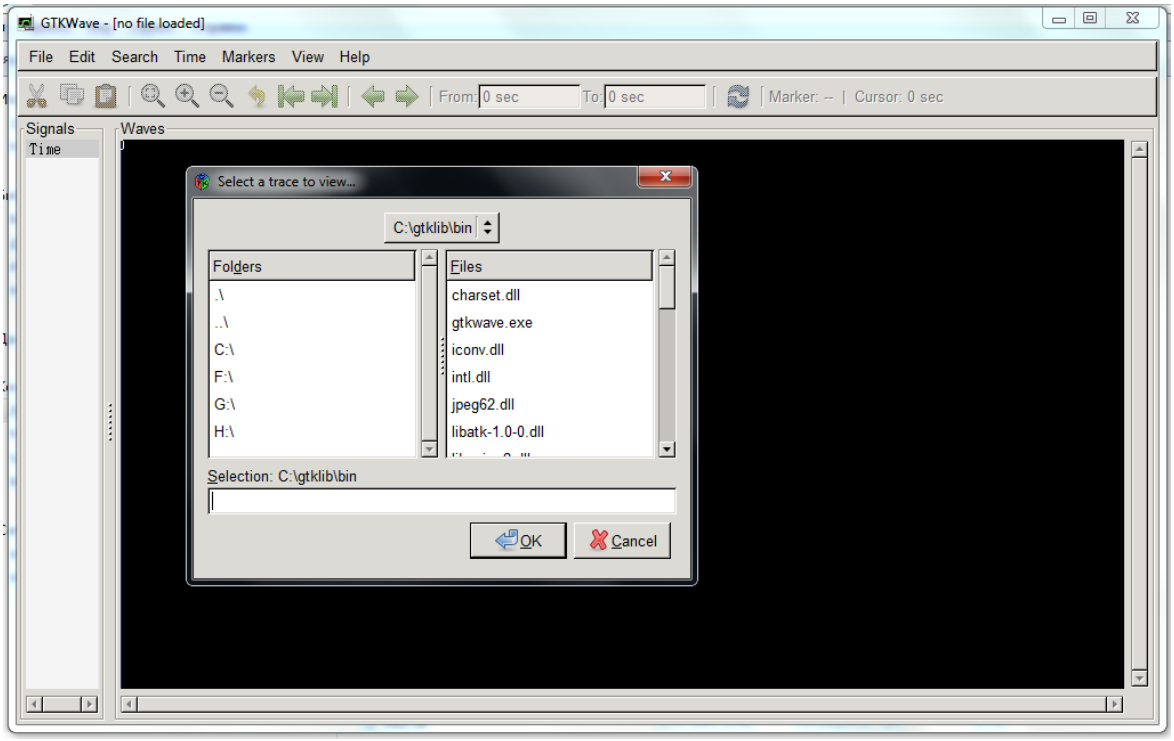


Рис. 2.49. Рабочее окно GTKWave

6. Копируем с сайта ACS SystemC Examples: First Experiment (Hello World) тестовый файл hword2.cpp и вставляем в папку src проекта Syst-test в среде Eclipse. Выполняем компиляцию и запускаем решение (рис. 2.50).

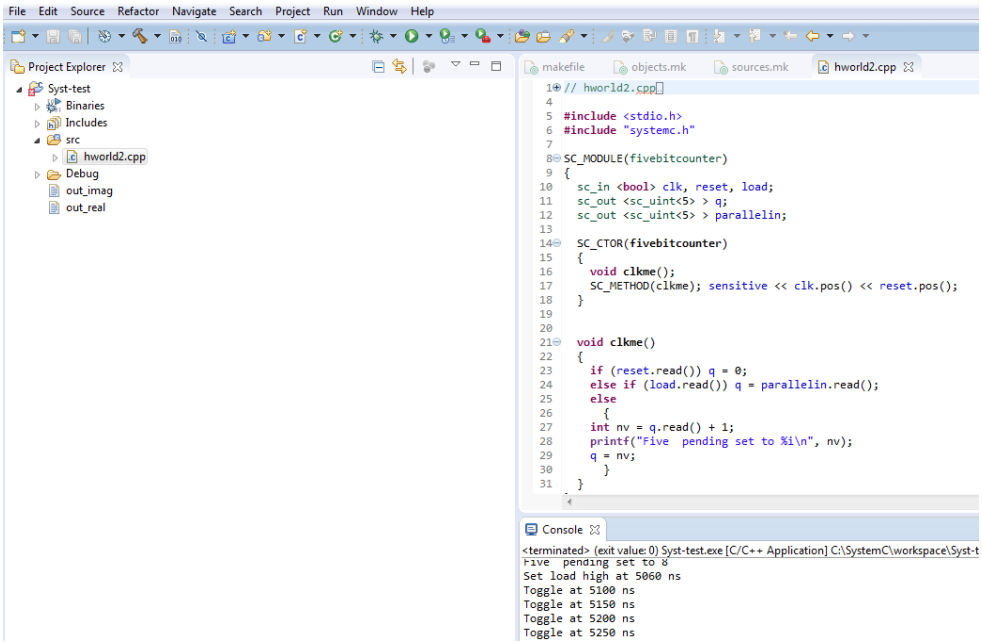


Рис. 2.50. Решение для hword2.cpp

В консоли наблюдаем сообщения о переключениях через 50 нс.

7. Открываем File>Open New Tab и в директории C:\SystemC\workspace\Syst-test находим файл trace.vcd (рис. 2.51).

Чтобы упростить поиск таких файлов для повторных просмотров, можно создать папку C:\VC-GTK-Trace и скопировать в нее файлы vcd*.

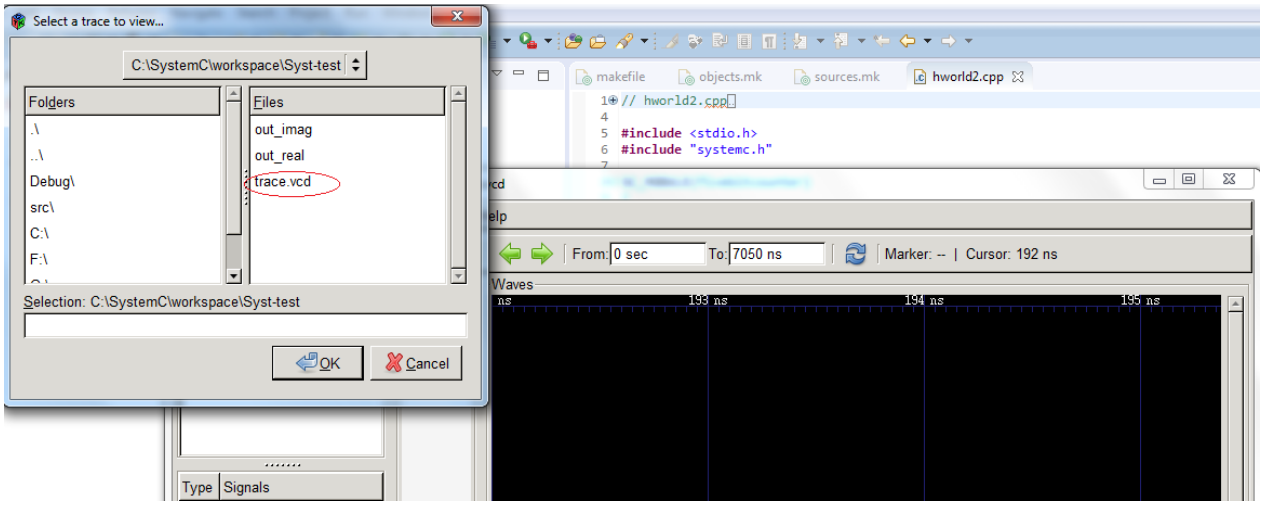


Рис. 2.51. Файл trace vcd

8. Нажимаем в окне SST на SystemC и нижнем окне видим выведенные типы сигналов и их названия. Последовательно выделяем каждый сигнал и нажимаем Insert.

В окне Waves появляются сначала непонятные зеленые полосы и импульсы. Слева в окне Signals указаны названия сигналов.

В меню Time>Zoom выделяем Zoom Best Fit (рис. 2.52) и наблюдаем диаграммы сигналов в оптимальном масштабе времени (рис. 2.53).

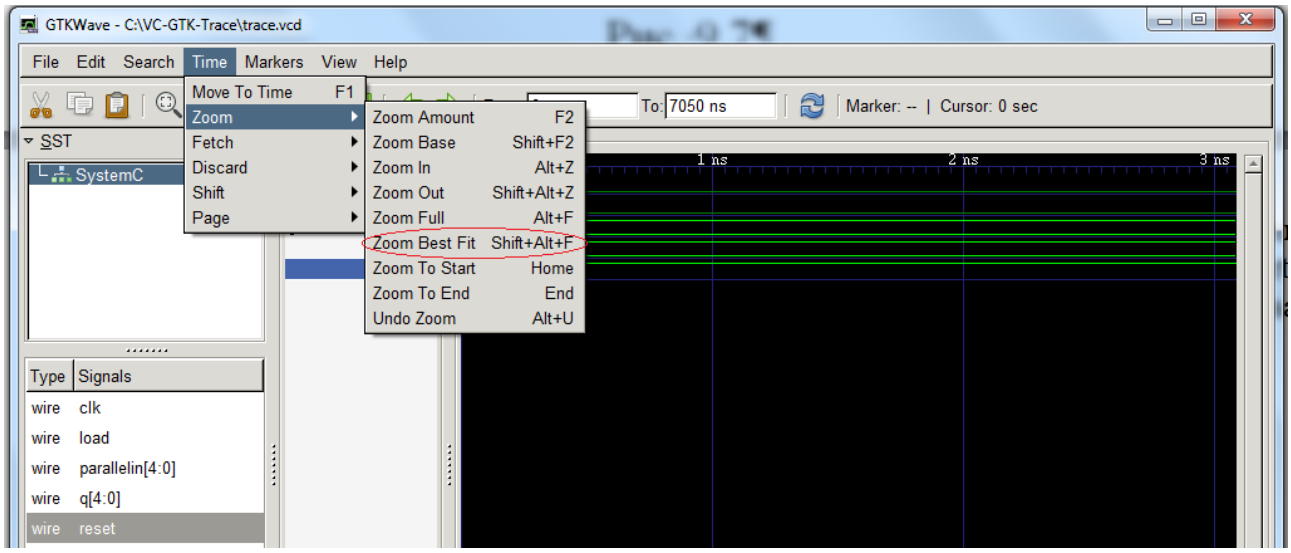


Рис. 2.52. Диаграммы сигналов

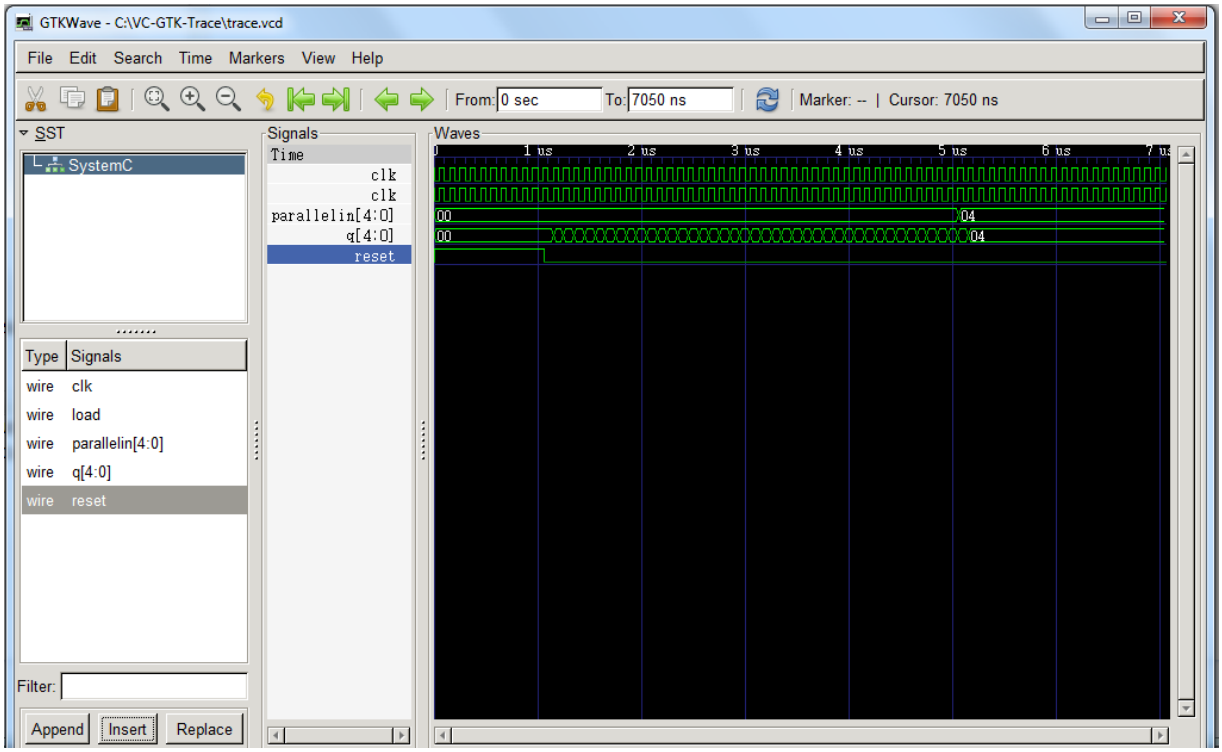


Рис. 2.53. Оптимальный масштаб диаграмм

2.4. Установка SystemC-2.3.1 в операционной системе Ubuntu 16.04

Ubuntu построен на основе Linux, который является членом Unix семьи. Unix - один из старейших типов операционных систем, и вместе с

Linux обеспечила надежность и безопасность для профессиональных приложений. Современные автомобильные компьютеры обычно работают на Linux. Ядро Linux лучше всего описывается как ядро - почти мозг - операционной системы Ubuntu. Ядро Linux является контроллером операционной системы; и отвечает за распределение памяти и времени процессора. Его также можно рассматривать как программу, которая управляет всеми приложениями на самом компьютере.

Linux был разработан с нуля с помощью средств безопасности и оборудования совместимости, и в настоящее время является одним из самых популярных Unix-based операционных систем. Одним из преимуществ Linux является то, что он невероятно гибкий и может быть сконфигурирован для работы практически на любом устройстве - от самого маленького микро-компьютеров и мобильных телефонов для крупнейших суперкомпьютеров.

Ubuntu 16.04 одна из новейшая и самая продвинутая версия Ubuntu OS, получившая кодовое имя Xenial Xerus от основателя Canonical Марка Шаттлворта. Ubuntu 16.04 LTS (Xenial Xerus) использует версию ядра Linux с долгосрочной поддержкой - версия 4.4.

Ubuntu имеет собственный эффективный компилятор Linux-64, пригодный для C++ и SystemC. Поэтому установка SystemC в среде Ubuntu имеет свои обобенности.

Мы исходим из того, что Ubuntu установлена на Вашем компьютере. Если нет, рекомендуем Вам изучить в Интернете сайты с указаниями как это сделать.

Так как установка SystemC выполняется с использованием терминала, кратко рассмотрим наиболее полезные команды Ubuntu.

Чтобы открыть терминал, наберите Ctrl-Alt-t.

2.4.1. Полезные команды для работы в терминале Ubuntu

Системные команды:

`man` - эта команда выводит справочную информация по нужной вам команде, такую как синтаксис, ключи, описание и т.д.

Синоним info.

`sudo` - переход в режим суперпользователя (после первой команды `sudo` необходимо ввести пароль пользователя).

Для запуска в терминале команды с правами администратора просто наберите перед ней `sudo`. Пример: `sudo reboot`

`reboot` - перезагрузка системы

`poweroff` - выключение компьютера

`reset` - очищает окно терминала, работает даже если вы потеряли курсор

`passwd` - позволяет пользователю поменять свой пароль, а суперпользователю - поменять пароль любого
зарегистрированного в
системе пользователя
`free` - получить информацию об оперативной памяти (всего, занято, свободно, в swap).

Команды для работы с файлами

`pwd` – показать текущий каталог
`cd` - переход в заданную папку
`ls` - показывает список файлов текущей папки, с ключом -l показывает дополнительные сведения о файлах
`cp` - копирование файлов/папок
`mv` - перемещение файлов/папок
`rm` - удаление файлов/папок, с ключем -R удаляет и все вложенные папки
`mkdir` - создать папку
`rmdir` - удаление пустой папки
`chmod` - изменить права доступа к файлу.

Команды для работы с пакетами

Это одна команда (утилита) с разными ключами.
`apt-get update` - обновление информации о пакетах из репозиториев
`apt-get upgrade` - обновление всех пакетов
`apt-get clean` - очищает локальный репозиторий, т.е. удаляет всё, что вы ранее скачивали. Очень полезно иногда
прогонять для очистки диска.
`apt-get autoremove` - удаление ранее скачаных, но более ненужных пакетов
`apt-get remove` - удаление пакета из системы, с сохранением его конфигурационных файлов
`apt-get purge` - удаление пакета со всеми зависимостями
`apt-get install` - установка пакета.

Для перемещения библиотек в каталог /usr/lib/ выполнить:

```
$  sudo mv /usr/local/systemc-2.3.1a/lib-linux64/libsystemc.so
```

```
/usr/lib/libsystemc.so
```

2.4.2. Установка SystemC-2.3.1a

1. С сайта Accellera.com надо загрузить программу SystemC-2.3.1a. и распавковать программу в корневую папку / «Компьютер».

2. Открыть терминал Ubuntu и выполнить команду перехода в заданную папку:

```
$ cd systemc-2.3.1a
```

Проверить файлы, содержащиеся в этой папке командой:

```
$ ls
```

Большинство установок Linux имеют необходимые инструменты по умолчанию, просто надо убедиться, что они установлены или установить их:

```
$ sudo apt-get install make build-essential
```

Чтобы проверить, что все готово, используйте команду:

```
which g++
```

Ответом будет: `/usr/bin/g++`

Затем надо установить несколько переменных окружения:

```
$ export CXX=g++
```

Если у вас есть пользовательский скомпилированный GCC, у Вас есть папка `/use/local/bin/gcc` и компилятором `g++` будет по умолчанию. Вероятно, это старая версия для компиляции. Тогда Вам надо более конкретно установить в системе пути GCC и выполнить команды:

```
$ export CXX=/usr/bin/g++
$ export CC=/usr/bin/gcc
```

Затем создайте директорию, где Вы будете компилировать библиотеки перед их установкой:

```
$ mkdir objdir
```

Войдите в эту директорию:

```
$ cd objdir
```

Создайте папку в директории `/usr/local` , в которую будет выполнена установка.

```
$ sudo mkdir /usr/local/systemc-2.3.1
```

Так как создана папка objdir, выполните команду:

```
$../configure -prefix=/usr/local/systemc-2.3.1
```

Эта команда настраивает папку для установки и генерирует среди прочих makefifes.

Несколько последних строк выглядят так:

```
config.status: creating examples/tlm/Makefile
config.status: creating
docs/sysc/doxygen/Doxyfile
config.status: creating docs/tlm/doxygen/Doxyfile
config.status: executing depfiles commands
config.status: executing libtool commands
```

Предполагая, что конфигурация прошла гладко, компилируем библиотеку:

```
$ make
```

Устанавливаем библиотеку в систему:

```
$ sudo make install
```

После этой команды Вы можете проверить, что библиотека правильно установлена:

```
$ ls /usr/local/systemc-2.3.1/lib-linux64/
libsystemc-2.3.0.so  libsystemc.a  libsystemc.la
libsystemc.so
```

Для обоих систем systemc-2.3.0 и systemc-2.3.1 названия библиотек одни и те же: `libsystemc-2.3.0.so`

Примечание: для 32-битных систем папке библиотеки `/usr/local/systemc-2.3.1/lib-linux/` надо ввести во-время использования табло для пути автокомпиляции.

Чтобы сконфигурировать библиотеку со стандартным путем к библиотеке Linux укажите путь к библиотеке, который упрощает этап сборки проекта, введите команду символической ссылки на файл или директорию:

```
$ sudo ln -s /usr/local/systemc-2.3.1/lib-
linux64/libsystemc-2.3.0.so /usr/lib/libsystemc-
2.3.1.so
```

Можно проверить правильность пути:

```
ls -l /usr/lib/libsystemc-2.3.1.so
lrwxrwxrwx 1 root root 56 Feb 21 12:25
/usr/lib/libsystemc-2.3.1.so -> /usr/local/systemc-
2.3.1/lib-linux64/libsystemc-2.3.0.so
```

Эта проблема пути к библиотеке в последнее время была частой и решение в том, чтобы добавить конфигурационный файл под `/etc/ld.so.conf.d/` следующим образом:

```
$ sudo gedit /etc/ld.so.conf.d/systemc.conf
```

Добавьте информацию о пути к этому файлу, i.e `"/usr/local/systemc-2.3.1/lib-linux/"` или

```
"/usr/local/systemc-2.3.1/lib-linux64/"
```

в зависимости от вашей системы.

Затем запустите:

```
$ sudo ldconfig
```

2.4.3. Установка JAVA в Ubuntu

Напомним, что для работы Eclipse требуется установка программы JAVA.

Для начала, обновим репозитории в системе.

```
$ sudo apt-get update
```

Проверим какая версия java в системе и установлена ли она вообще.

```
$ java -version
```

Если в вашей ОС, нет этих пакетов, вы увидите, что-то похожее...

Программа 'java' может находится в следующих пакетах:

```
default-jre
gcj-4.9-jre-headless
gcj-5-jre-headless
openjdk-7-jre-headless
gcj-4.8-jre-headless
openjdk-6-jre-headless
openjdk-8-jre-headless
openjdk-9-jre-headless.
```

Выполните команду:

```
$ apt install <selected package>.
```

После выполнения нажать Y

2.4.4. Установка Eclipse в Ubuntu

С сайта Eclipse.org загрузите Eclipse IDE для разработчиков C/C++. Распакуйте архив и запустите программу.

Создайте новый C++ проект, выберите компилятор Linux-GCC.

Создайте папку `src` для файлов и файл `main.cpp`.

Откройте папку «Свойства проекта» и сделайте установки:

Для вложенный файлов (рис. 2.54) введите:

```
/usr/local/systemc-2.3.1a/include
```

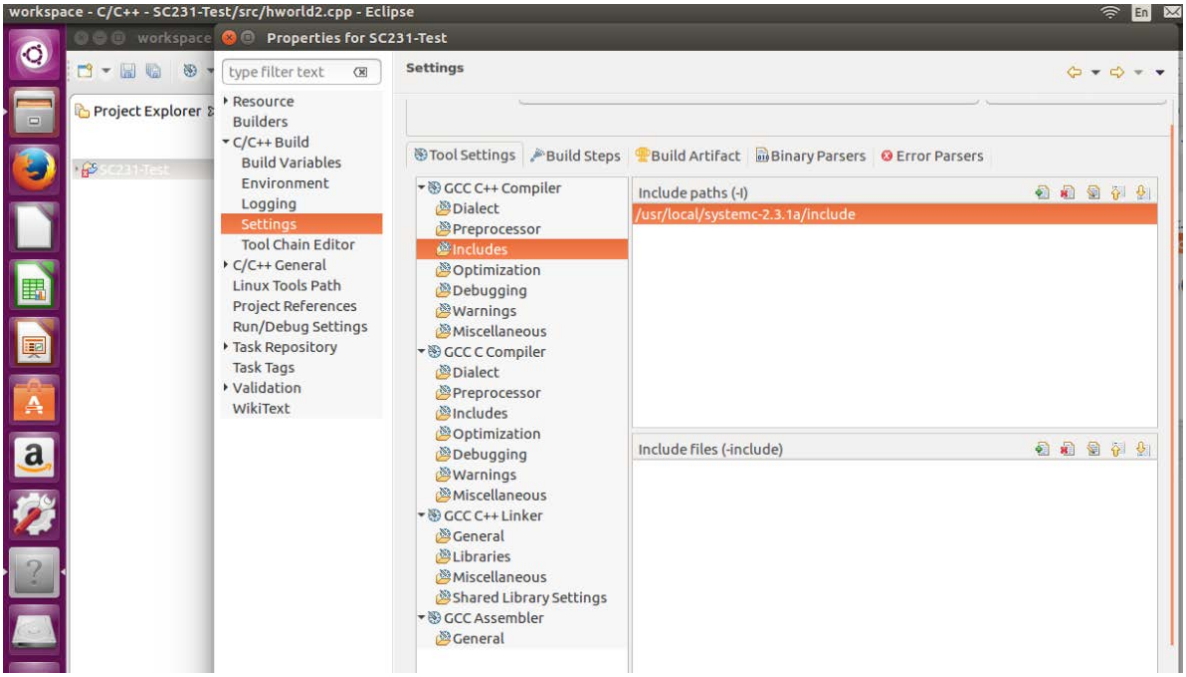


Рис. 2.54. Вложенные файлы

Для библиотек Linker (рис. 2.55) введите:

```
Libraries = systemc;
Path=/usr/local/systemc-2.3.1a/lib-linux64/
```

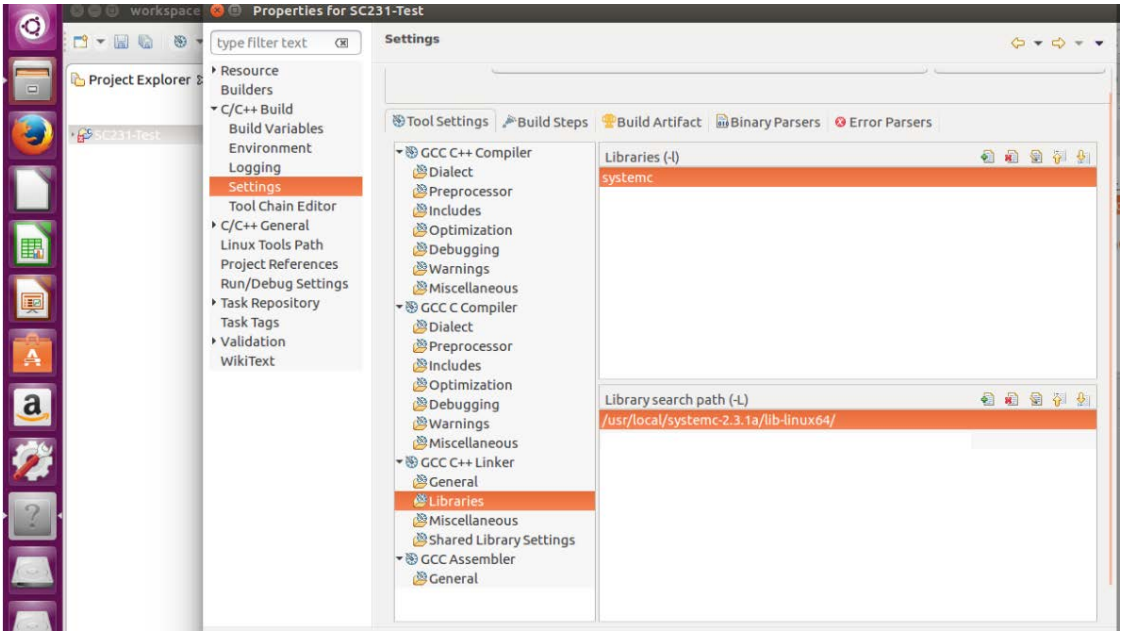


Рис. 2.55. Библиотеки линкера

Проверим работу с файлом hworld2.cpp (рис. 2.56)

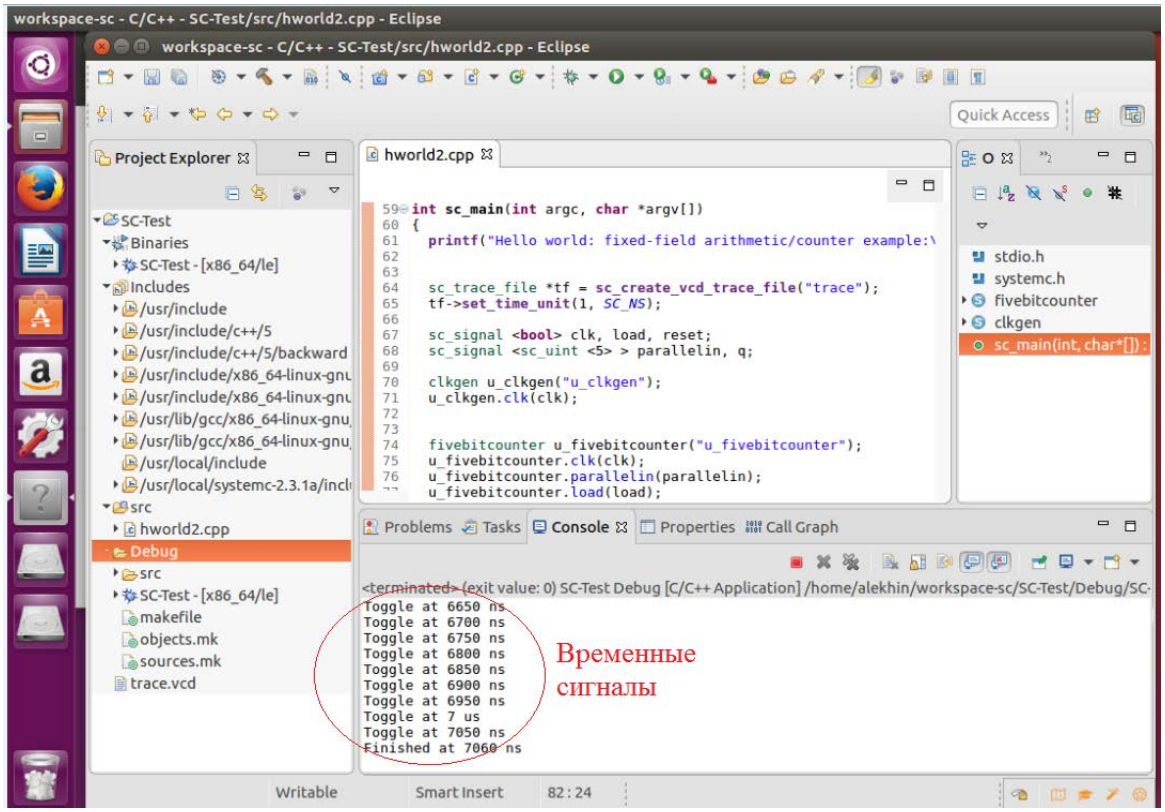


Рис. 2.56. Решение для hworld2.cpp

2.4.5. Установка GTKWave

Подлючиться к Интернету и выполнить в терминале:

```
Sudo apt-get update
Sudo apt-get install gtkwave
```

В поисковике находим GTKWave.

Запуск файла VCD из workspace открывает GTKWave (рис. 2.57):

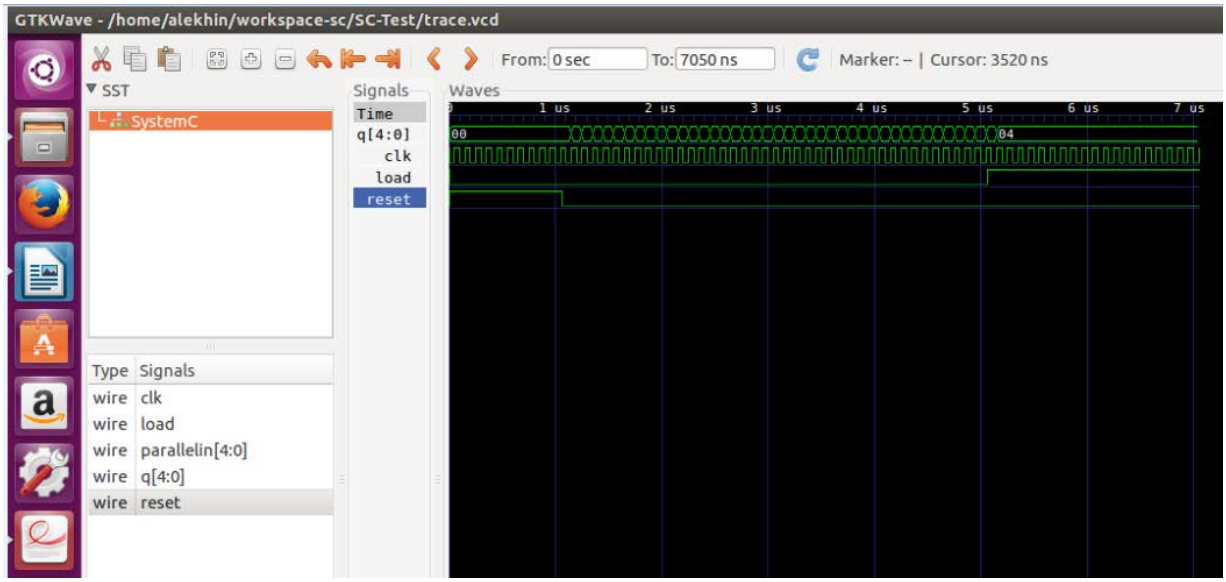


Рис. 2.57. Рабочее окно GTKWave в Ubuntu

2.5. Об установке SystemC в Microsoft Visual Studio

Разработчики SystemC предоставляют возможность для работы с программой в Microsoft Visual Studio C++ (далее MSVC++). Дистрибутив SystemC содержит специальные файлы для запуска MSVC++ и компиляции библиотек с использованием компилятора GCC.

Настройка проектов в MSVC++ достаточно трудоемка и требует многочисленных установок свойств проекта. После многих попыток автору удалось добиться успеха. В Приложении Б подробно описано, как это сделать.

Глава 3. Основы языка SystemC-2.3.1

3.1. SystemC – надстройка к языку C++

SystemC обращается к моделированию программного и аппаратного обеспечения, используя C++. Диаграмма (рис. 3.1) иллюстрирует основные компоненты SystemC, которые базируются на стандартном языке C++. Поскольку C++ уже решает большинство задач программного обеспечения, не удивительно, что SystemC фокусируется прежде всего на проблемах, связанных как с программным обеспечением, так и с аппаратной реализацией. Основной областью применения SystemC является разработка электронных систем, но SystemC применяется к неэлектронным системам.

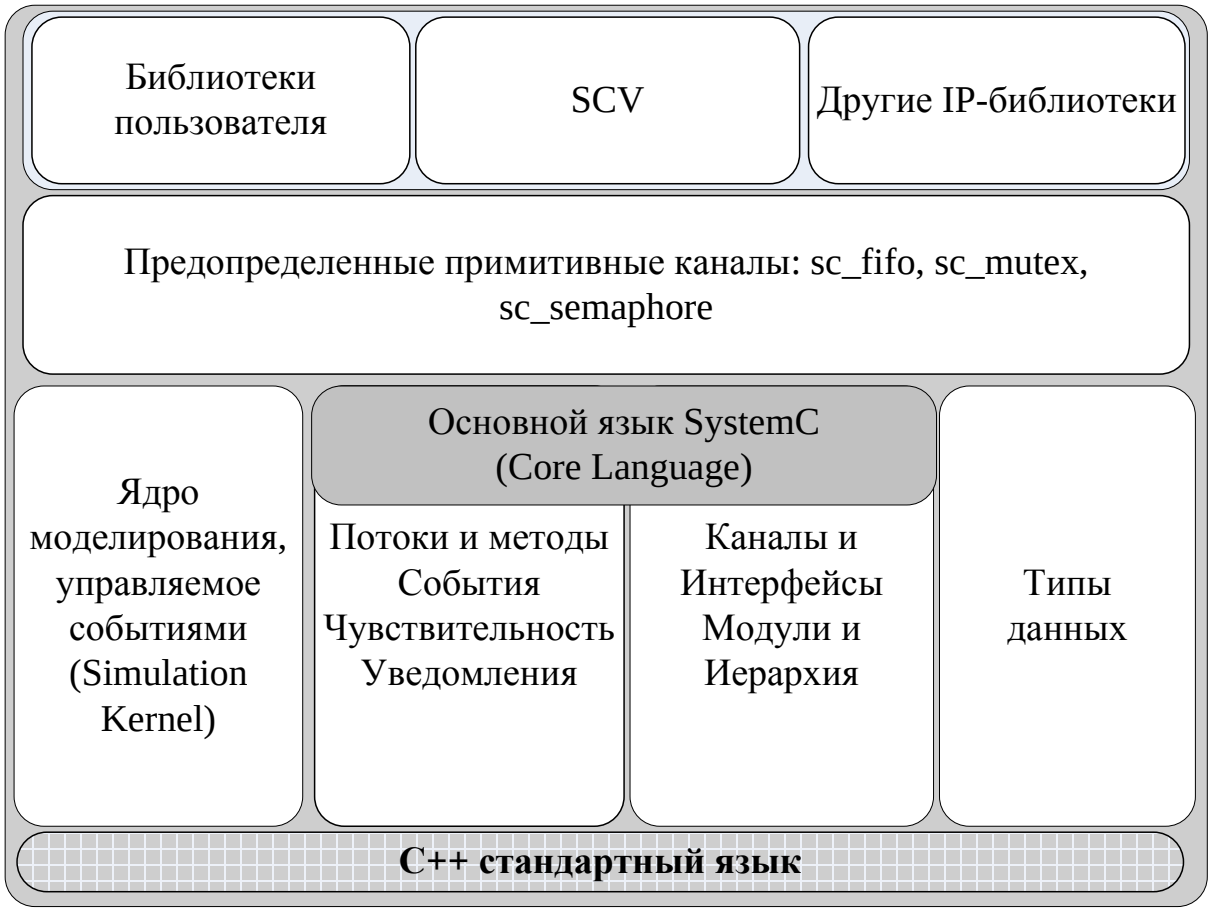


Рис.3.1. Архитектура и главные компоненты языка SystemC

Проект на языке SystemC состоит из набора файлов .cpp и .h. Для того, чтобы полноценно работать, не требуется какая-либо среда разработки, поддерживающая данный язык. Достаточным условием является наличие компилятора C++, библиотеки SystemC и среды

разработки для языка C++. Мы будем работать в среде Eclipse CDT и Visual Studio. В качестве компилятора, как правило, будет использован Cygwin.

3.2. Ядро моделирования (Kernel)

Надстройка к языку C++ (рис. 3.1) включает в себя основные компоненты языка, с которыми мы будем постепенно знакомиться.

Начнем с ядра моделирования Simulation Kernel.

Имитатор SystemC имеет 3 главные фазы работы: разработка, выполнение и постобработка. Выполнение всех операторов до конструкции sc_start() есть фаза разработки. На этом этапе происходит инициализация структур данных и подготовка к следующей фазе выполнения. Фаза выполнения передаёт управление ядру (Kernel) моделирования SystemC, которое управляет работой всех процессов и создаёт иллюзию параллельности их выполнения. Постобработка связана с удалением всех созданных структур данных, освобождением памяти и завершением этапа моделирования.

Работа ядра моделирования и главная программа

Принцип работы ядра моделирования SystemC схож с языками VHDL и Verilog. Если обратиться к Verilog и VHDL, проходит некоторое время между инициализацией кода и началом моделирования. В SystemC, также как и в C/C++, есть строго определённая точка входа в программу. В случае SystemC это sc_main().

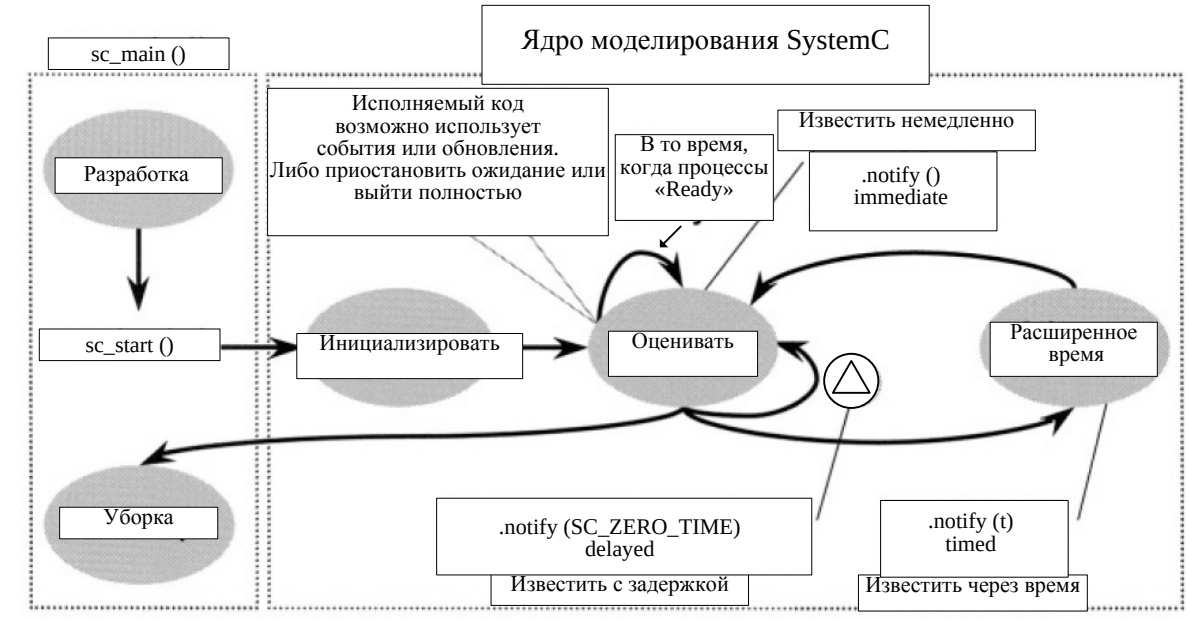


Рис. 3.2. Работа ядра моделирования SystemC

Исполнение команд перед вызовом функции в sc_start () известно как этап разработки. Эта фаза характеризуется инициализацией структур данных, установлением соединений, а также подготовкой ко второму этап - выполнению. Контроль фазы выполнения передан ядру моделирования SystemC, которое дирижирует выполнением процессов, чтобы создать иллюзию параллелизма.

Иллюстрация на рис. 3.2 должна выглядеть очень хорошо знакомой тем, кто изучали ядра моделирования Verilog и VHDL. После команды sc_start () все процессы моделирования (за минусом несколько исключений) вызывается случайным образом во время инициализации. После инициализации процесс моделирования запускается, когда происходит событие, к которому процесс чувствителен. Несколько процессов моделирования могут начаться в один и тот же момент времени в симуляторе. Из-за этого случая, все процессы моделирования оцениваются и затем их выходы обновляются. Оценки с последующим обновлением называются *дельта-цикл*. Если нет никаких дополнительных процессов моделирования, которые требуется оценить в настоящее время (в результате обновления), то время моделирования продвигается вперед. При этом, если не требуется запускать дополнительные процессы моделирования, моделирование заканчивается.

Этот краткий обзор работы ядра моделирования предназначен для того, чтобы дать понимание остальной части книги. Позже эта схема будет использоваться снова, чтобы объяснить важные тонкости. Очень важно понять как функционирует ядро, чтобы полностью понять язык SystemC. SystemC LRM (Справочное руководство) определяет поведение ядра SystemC моделирования.

3.3. Состав ядра языка SystemC (Core Language)

Ядро языка SystemC (Core Language), как показано на рис. 3.1, кроме ядра моделирования включает в себя следующие компоненты: модули, порты, процессы, события, интерфейсы, каналы. Эти компоненты оперируют с различными типами данных, как правило, аналогичными языку C++.

Основные понятия ядра языка SystemC классифицируются по следующей терминологии:

Основные термины

Method	Метод C ++, т.е. функция-член класса.
Module	Конструктивный субъект, который может содержать

	процессы, порты, каналы, и другие модули. Модули позволяют выразить структурную иерархию.
Interface	Интерфейс предоставляет набор объявлений методов, но не дает никаких реализаций метода и нет полей данных.
Channel	Канал реализует один или несколько интерфейсов, а также служит в качестве контейнера для функциональных возможностей связи.
Port	Порт представляет собой объект, через который модуль может получить доступ к его каналу интерфейс. Но модули могут также получить доступ к интерфейсу канала напрямую.
Primitive Channel	Примитивный канал является атомарным, то есть, он не содержит процессы или модули, и он не может непосредственно получить доступ к другим каналам.
Hierarchical Channel	Иерархический канал представляет собой модуль, то есть, он может содержать процессы и другие модули, и он может непосредственно получить доступ к другим каналам.
Event	Событие. Процесс может приостановливаться событием или быть чувствительным к одному или нескольким событиям. События позволяют возобновить и активизировать процессы.
Sensitivity	Чувствительность процесса определяет, когда этот процесс будет возобновлен или активирован. Процесс может быть чувствительным к набору событий. Всякий раз, когда одно из соответствующих событий инициируется, процесс возобновляется или активируется.
Static Sensitivity	Чувствительность процесса объявляется статически, т.е. заявляется во время разработки и не может быть изменена после начала моделирования. Так называемый список чувствительности используется для определения статического набора событий.

Dynamic Sensitivity	Чувствительность процесса может быть изменена во время моделирования.
IMC	Метод вызова интерфейса
RPC	Удаленный вызов процедур

Терминология процессов

Процессы играют центральную роль в SystemC. Они описывают функциональные возможности системы и позволяют выразить параллелизм в системе. Процессы содержатся в модулях и они имеют доступ к интерфейсам внешнего канала через порты модуля. Существуют различные типы процессов и различные способы активации процессов. Перед погружением в детали, объясним термины, связанные с процессами.

Thread	Поток. SystemC имеет свой собственный поток исполнения, но он не является упреждающим.
Automatically activated	Некоторые методы модуля (процессы) активируются автоматически, когда происходят события, к которым процессы чувствительны.
Explicitly activated	Некоторые методы модуля, которые должны быть вызваны явно другим кодом, чтобы активироваться.
wait()	Метод, который приостанавливает выполнение потока. Аргументы, передаваемые wait() определяют, когда выполнение потока возобновляется.
Ok to call `wait()`	SC_METHODs и код, который они называют, не могут вызвать wait(), так как они не имеют свой собственный поток исполнения. SC_THREADs и код, который они называют, могут вызвать ожидание ().
SC_THREAD	Модуль-способ, который имеет свой собственный поток исполнения, и который может вызвать код, который вызывает ожидание (). SC_THREADs автоматически активируются. Также известный как процесс потока.
SC_METHOD	Модуль - метод, который не имеет свой собственный

	поток исполнения, и который не может вызвать код, который вызывает wait(). SC_METHODs автоматически активируется. Также известный как процесс метода.
SC_CTHREAD	Модуль - метод, который имеет свой собственный поток исполнения, и который в его списке чувствительности имеет только положительное или отрицательное событие по фронту тактового импульса. Он может вызвать код wait() с ограниченным списком аргументов. SC_CTHREADs активируется автоматически. Также известный как процесс с тактовой частотой.

Следующей более высокой надстройкой являются понятия элементарных каналов.

На вершине архитектуры языка SystemC добавлены более специфические модели вычислений, библиотеки проектирования, руководства по моделированию, методология проектирования, которые полезны при проектировании.

Нижний слой подчеркивает то, что SystemC создан полностью на языке C++ . Это означает, что любая программа, написанная на SystemC, может быть откомпилирована компилятором C++, чтобы получить исполняемую программу.

3.4. Инициализация процесса

В SystemC планировщик будет выполнять все потоковые процессы и все процессы методов во время фазы инициализации моделирования. Чтобы предотвратить действия планировщика от выполнения процесса потока или процесса метода во время инициализация фазы моделирования, вы можете использовать функцию dont_initialize (). Функция применяется в конце объявления процесса. Например,

```
SC_MODULE( my_module )
{
// порты
sc_in_clk clk;
// процессы
void proc_a();
void proc_b();

// конструктор
SC_CTOR( my_module )
```

```
    {
    SC_THREAD( proc_a );
    sensitive << clk.pos();
    dont_initialize(); /* не инициализировать процесс
proc_a*/
    SC_METHOD( proc_b );
    sensitive << clk.neg();
    dont_initialize(); /* не инициализировать процесс
proc_b*/
    }
    };
```

3.5. Модель времени в SystemC

Так как время симуляции дискретно, существует единица разрешения по времени (time resolution), минимальный квант времени, который должен быть задан. По умолчанию время разрешения составляет 10^{-12} с (одну пикосекунду) и пользователь имеет опцию установки времени разрешения sc_set_time_resolution().

В библиотеке SystemC существует тип данных sc_time, для того чтобы измерять время в процессе моделирования. У времени есть две составляющие: числовое значение и размерность. Языком поддерживается измерение времени в секундах, миллисекундах, микросекундах, наносекундах, пикосекундах, фемтосекундах.

Соответствующие спецификаторы времени следующие:

```
SC_SEC // секунды
SC_MS // миллисекунды
SC_US // микросекунды
SC_NS // наносекунды
SC_PS // пикосекунды
SC_FS // фемтосекунды
```

Объявление переменных времени выглядит следующим образом:

```
sc_time имя_переменной (числовой значение,
спецификатор);
sc_time t_Period(10, SC_NS);
```

Это означает, что объект `t_Period` представляет 10 нс.

Временные объекты:

```
sc_set_time_resolution(10,SC_PS);
sc_time t2(3.1416, SC_NS);
```

означают, что будет установлено временное разрешение 10 пс, а время t2 будет установлено 3.1416 нс.

Над переменными типа sc_time SystemC позволяет производить операции сложения, вычитания, масштабирования и др. В библиотеке определена константа SC_ZERO_TIME, которая соответствует времени 0.

3.6. Модули SC_MODULE

Основной единицей проектирования является модуль SC_MODULE.

Сложные системы состоят из множества независимо функционирующих компонентов. Эти компоненты могут представлять собой аппаратные средства, программное обеспечение или любой объект. Компоненты могут быть большими или маленькими. Компоненты часто содержат иерархии более мелких компонентов. Самые маленькие компоненты представляют поведение и состояние. В SystemC мы используем концепцию, известную как SC_MODULE для представления компонентов.

ОПРЕДЕЛЕНИЕ: Модуль – это конструктивный субъект, который может содержать процессы, порты, каналы, и другие модули. Модули позволяют выразить структурную иерархию.

Модуль SystemC является самым маленьким контейнером функциональности с состоянием, поведением и структурой для иерархического подключения.

Модуль SystemC соответствет определению класса C++. Как правило, макрос SC_MODULE используется для объявления класса:

```
#include <systemc.h>
SC_MODULE(Adder) {
//Тело модуля
//порты, процессы, внутренние данные и т.д.
SC_CTOR(Adder){
//Тело конструктора
//объявление процессов, чувствительностей и т.д.

}
};
```

SC_MODULE – это простой макрос C++ и на языке C++ его можно определить так:

```
#define SC_MODULE (module_name)
struct module_name: public sc_module
```

В рамках этого производного класса модуля, разнообразные элементы составляют MODULE BODY:

Порты;

Отдельные интерфейсов каналов;
Отдельные данные;
Отдельные модули или подмодули;
Конструктор;
Деструктор;
Процессы;
Вспомогательные функции.

Из них необходимым является только конструктор. Однако, чтобы иметь любое полезное решение, вы должны иметь либо процесс или sub-design. Сначала мы посмотрим на конструктор, а затем на простой процесс. Эта последовательность позволяет нам разобраться с базовым примером минимального дизайна.

Модуль в SystemC – это базовый элемент, включающий в себя процессы (processes) и другие модули.

МОДУЛЬ является старшим в иерархии элементов SystemC.

Наличие МОДУЛЯ позволяет строить SystemC-модели в соответствии с имеющимися у разработчиков аппаратуры представлениями (документацией) об архитектуре и функционировании будущего изделия. Обычно каждый отдельно взятый МОДУЛЬ представляет в модели функционально законченный узел разрабатываемого изделия.

МОДУЛИ объявляются с ключевым словом SC_MODULE.

На рис. 3.3 изображён модуль, который включает в себя несколько *процессов*.

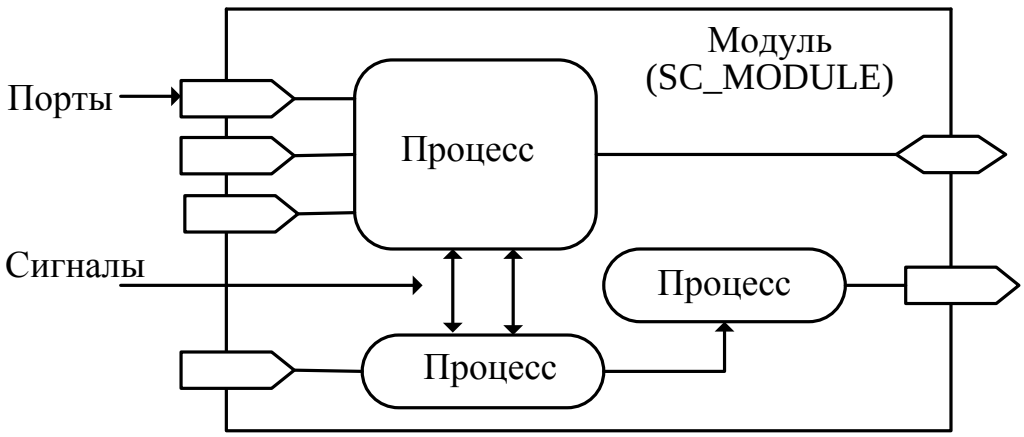


Рис. 3.3. Модуль, содержащий несколько процессов

Структура модуля

```
// Заголовочный файл
SC_MODULE(module_name) {
Объявление портов
Объявление локальных каналов
```

```
Объявление переменных
Объявление процессов
Объявление других методов

Конкретизации модуля
SC_CTOR(module_name){
Регистрация процесса
Лист статической чувствительности
Инициализация переменных модуля
Связывание экземпляра модуля / канала}
} ;
```

3.6.1. Порты модулей

Порты модуля передают данные к процессам модуля и из него во внешний мир, как в Verilog и VHDL. Вы объявляете направление порта как внутри, так и снаружи. Вы также объявляете тип данных порта как любой тип данных C++, тип данных SystemC или пользовательский тип. Типы портов:

In: входные порты

Out: выходные порты

Inout: двунаправленные порты

Режимы порта `sc_in, sc_out и sc_inout` предопределены библиотекой классов SystemC.

Синтаксис:

Переменная типа sc_direction;

Далее:

`Port_direction`: один из `sc_in, sc_out, sc_inout`

`Type`: тип данных;

`Variable`: Имя действительной переменной

```
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in_clk     clock ;      /* Входной тактовый
сигнал проекта */
  sc_in<bool>   reset ;      /* активный высокий,
синхронный вход сброса */
  sc_in<bool>   enable;      /* Активный высокий
разрешающий сигнал для счетчика */
  sc_out<sc_uint<4> > counter_out; /* 4-битный
векторный выход счетчика */
```

```
  // Остальная часть тела
}
```

3.6.2. Сигналы модуля

Порты используются для связи вне модуля. Для связи внутри модуля SystemC мы используем сигналы. Сигналы подобны проводам в Verilog. Сигналы могут быть любых допустимых типов данных.

Сигнал также используется для соединения двух портов модулей в родительском модуле. Допустим, у нас есть два дочерних модуля A и B, эти два модуля используются в родительском модуле C, тогда провода используются для соединения портов модуля A и B друг с другом.

Синтаксис:

Переменная типа `sc_signal`;

Пример:

`sc_signal`: зарезервированное слово

`Type`: Тип данных

`Variable`: Имя действительной переменной

```
#include "systemc.h"

SC_MODULE (counter) {
  sc_signal <bool>         reset ;
  sc_signal <bool>         enable;
  sc_signal <sc_uint<4> > counter_out;

  // Остальная часть тела
}
```

Типы данных и операторы SystemC подробно будут рассмотрены позже. Для справки смотрите раздел 3.18.

3.6.3. Создание экземпляров модулей

Для создания экземпляров модулей (Instanciating) в SystemC так же, как в Verilog, соблюдаются те же правила. Модуль SystemC, подобный Verilog, позволяет использовать два способа соединения портов.

По положению;

По имени.

По положению

Здесь порядок должен совпадать. Обычно не рекомендуется подключать порты по положению. Это может вызвать проблемы при отладке (например: трудно найти порт, вызывающий ошибку компиляции), когда новый порт добавляется или удаляется. Ниже приведен пример подключения портов по позиции.

```
include "systemc.h"
#include "first_counter.cpp"

int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  /* Подключение тестируемого устройства (DUT)
(тестируемый проект)*/
  first_counter counter("COUNTER");
    // Здесь порты соединены по положению
    counter(clock,reset,enable,counter_out);

  // Остальная часть тестового стенда

  return 0;// Завершить симуляцию
 }
```

По имени:

Здесь имя должно совпадать с исходным модулем, порядок не имеет значения. Ниже приведен пример соединения портов по имени.

```
#include "systemc.h"
#include "first_counter.cpp"

int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Подключить DUT
  first_counter counter("COUNTER");
    // Здесь порты связаны по имени
    counter.enable(enable);
    counter.reset(reset);
    counter.clock(clock);
    counter.counter_out(counter_out);

  // Остальная часть тела

  return 0;// Завершить симуляцию
```

```
}
```

3.6.4. Внутренние переменные

SystemC позволяет использовать локальную переменную, так же Verilog или любой другой язык программирования. Локальные переменные могут быть любого допустимого C++ или SystemC типа или пользовательских типов.

```
#include "systemc.h"

SC_MODULE (local_variable) {
  sc_in_clk     clock ;    /* Входной тактовый
сигнал проекта*/
  sc_in<bool>   reset ;      /* активный высокий,
синхронный вход сброса*/
  sc_in<bool>   enable;      /* Активный высокий
разрешающий сигнал для счетчика*/
  sc_out<sc_uint<4> > counter_out; /* 4-битный
векторный выход счетчика*/

  //-------Локальные переменные здесь ----------
  sc_uint<4> count;

  //--------Код начинается здесь --------------
  /* Ниже функция реализует фактическую логику
счетчика*/
  void incr_count () {
    // Тело функции
  } // Конец функции incr_count

  // Конструктор для счетчика
  SC_CTOR(local_variable) {
    cout<<"Executing new"<<endl;
    SC_METHOD(incr_count);
    sensitive << reset;
    sensitive << clock.pos();
  } // Конец конструктора

}; // Конец модуля счетчика
```

Фактическая функциональность модуля SystemC реализована в процессах. Процесс может быть либо сделан чувствительным к уровню

модельной логики, либо может быть чувствительным к фронту последовательной логики модуля.

Процесс можно заставить работать вечно. Такие процессы называют потоками в SystemC. Или можно запускать процесс каким-либо событием Например, положительным фронтом тактового сигнала (posedge of clock).

Примечание. Сигналы чувствительного списка, которые запускают процесс / поток, должны быть портами и не могут быть сигналом или локальной переменной.

```
#include "systemc.h"

SC_MODULE(dff) {
  sc_in  <bool> din; // Ввод данных FF
  sc_in  <bool> clk; // Тактовый вход FF
  sc_out <bool> out; // Q выход FF

  void implement();  // Процесс, реализующий DFF
  SC_CTOR(dff) {
    SC_METHOD(implement);
    sensitive_pos << clk; /* Вызов реализуется () 
на каждом положительном фронте clk*/
  }
};
```

3.7. Конструктор SC_CTOR

КОНСТРУКТОР МОДУЛЯ SC_CTOR создает и инициализирует экземпляр МОДУЛЯ в симуляционном ядре SystemC. КОНСТРУКТОР создает внутренние структуры данных в ядре, которые используются МОДУЛЕМ и инициализирует их.

Синтаксис:

```
SC_CTOR(module_name)
{
тип ПРОЦЕССА(method_name)
}
```

Здесь:

SC_CTOR – Макрос;

module_name: Имя МОДУЛЯ, указанное в SC_MODULE («module_name»);

тип ПРОЦЕССА: Один из типов ПРОЦЕССОВ: SC_THREAD, SC_METOD, SC_CTHREAD;

method_name: Имя метода, реализованного в МОДУЛЕ.

Конструкторы модулей в SystemC реализованы таким образом, что имя экземпляра модулей и иерархия передается конструктору во время создания. Это помогает идентифицировать модуль при возникновении ошибок или при сообщении информации из модуля. В этом только разница между конструктором C++ и SystemC.

```
#include "systemc.h"

SC_MODULE(dff) {
  sc_in  <bool> din; // Вход данных FF
  sc_in  <bool> clk; // Тактовый вход FF
  sc_out <bool> dout;// Q выход FF

  void implement() {  // Процесс, реализующий DFF
    dout = din;
  }

  SC_CTOR(dff) {
    SC_METHOD(implement); /* Содержит процесс с
именем реализация*/
    sensitive_pos << clk; /* Выполнять действия()
для каждого положительного фронта clk*/
  }
};
```

3.8. Альтернативные конструкторы: SC_HAS_PROCESS

SystemC имеет и другой подход к созданию конструкторов. Альтернативный подход использует макрос .cpp SC_HAS_PROCESS. Вы можете использовать этот макрос в двух ситуациях. Во-первых, используйте SC_HAS_PROCESS, когда вам требуются конструкторы с аргументами, выходящими за пределы имени экземпляра строки, которая передается в SC_CTOR (например, для предоставления конфигурируемых модулей). Во-вторых, используйте SC_HAS_PROCESS, если вы хотите поместить конструктор в выполняемый файл (файл .cpp). Аргументы конструктора можно использовать, чтобы указать размеры подключенной памяти, диапазоны адресов для декодеров, глубину FIFO, делители тактовых импульсов, FFT глубину и другую конфигурационную информацию. Например, дизайн памяти может позволить выбор различных размеров памяти с аргументом:

```
My_memory instance("instance", 1024);
```

Чтобы использовать этот альтернативный подход, вызовите SC_HAS_PROCESS, а затем создайте обычные конструкторы. Применяется одна оговорка. Вы должны построить или ициализировать базовый класс модуля sc_module с помощью имени строки. Это требование является причиной того, что SC_CTOR нуждается в аргументе. Синтаксис этого стиля при использовании в заголовочном файле следующий:

```
//FILE: module_name.h
SC_MODULE(module_name) {
SC_HAS_PROCESS(module_name);
module_name(sc_module_name instname[,
other_args…])
: sc_module(instname)
[, other_initializers]
{
Тело конструктора
}
};
```

Синтаксис для использования SC_HAS_PROCESS в отдельной реализации (то есть при отдельной ситуации компиляции) аналогичен.

```
//FILE: module_name.h
SC_MODULE(module_name) {
SC_HAS_PROCESS(module_name);
module_name (sc_module_name
instname[,other_args…]);
};
//FILE: module_name.cpp
module_name::module_name(
sc_module_name instname[, other_args…])
: sc_module(instname)
[, other_initializers]
{
Тело конструктора
}
```

В предыдущих примерах, другие аргументы являются необязательными.

3.9. Процессы

ПРОЦЕСС - это встроенная функция (метод) класса SC_MODULE, которая запускается программой-планировщиком в ядре SystemC.

Синтаксис:

```
Объявление процесса:
```

```
void process_name();

Регистрация процесса:
SC_METHOD(process_name);
SC_THREAD(process_name);
SC_CTHREAD(process_name, clock_edge_reference);
SC_SLAVE(process_name, slave_port);
```

ПРОЦЕССЫ необходимы также для моделирования параллельного функционирования физических узлов изделия.

В ПРОЦЕССАХ декларируются методы `.write` (для записи данных в ПОРТЫ) и `.read` (для чтения данных из ПОРТОВ), а также внутренние сигналы (Integral Signals) – для передачи данных от одного ПРОЦЕССА к другому внутри МОДУЛЯ.

Для ПРОЦЕССА должен быть задан список чувствительности – перечень портов МОДУЛЯ, изменения на которых влияют на процесс, и список внутренних СОБЫТИЙ, наступление которых могут изменять ход его исполнения. Списки определяются в КОНСТРУКТОРЕ.

Процесс является основной единицей выполнения в SystemC. Всё выполнение кода инициируется из одного или нескольких процессов. Процессы, по всей видимости выполняться параллельно.

ОПРЕДЕЛЕНИЕ: процесс SystemC является методом или функцией-членом класса SC_MODULE, который вызывается планировщиком ядра моделирования в SystemC. Прототипом функции - члена для процесса SystemC является:

```
void PROCESS_NAME(void);
```

Процесс SystemC не принимает никаких аргументов и ничего не возвращает. Этот синтаксис делает его простым для вызова из ядра моделирования.

В SystemC предопределены три типа ПРОЦЕССОВ:

```
SC_THREAD; SC_METOD; SC_CTHREAD.
```

Процесс в SystemC объявляется в теле модуля и регистрируется как процесс внутри конструктора. Необходимо объявлять процесс как функцию void, не содержащую аргументов. При регистрации функции как SC_METHOD процесс, необходимо использовать конструкцию SC_METHOD, которая имеет один аргумент – имя процесса.

Пример описания процесса на SystemC:

```
SC_MODULE(my_module)
 {
sc_in<int> a;
sc_in<bool> b;
sc_out<int> x;
sc_out<int> y;
```

```
sc_signal<bool>c;
sc_signal<int> d;
void my_method_proc();
SC_CTOR(my_module)
{
 SC_METHOD(my_method_proc);
// Объявление списка чувствительных сигналов
}
};
```

Процессы реагируют на изменение сигналов, которые находятся в списке чувствительных входов. Возможно использование функций `sensitive()`, `sensitive_pos()` или `sensitive_neg()` или потоков `sensitive`, `sensitive_pos`, `sensitive_neg` при описании списка чувствительности (`sensitivity list`).

Для комбинаторной логики, список чувствительных входов приравнивает все входные порты (`input и inout ports`) и сигналы (`signals`) к входным сигналам процесса. Для реализации `level-sensitive` входов необходимо использовать метод `sensitive` так, как показано в примере:

```
SC_MODULE(my_module)
{ sc_in<int> a;
sc_in<bool> b;
sc_out<int> y;
sc_signal<bool>c;
void my_method_proc();
SC_CTOR(my_module)
{
SC_METHOD(my_method_proc);
// Объявление списка чувствительных сигналов
sensitive << a << c << d;
// Потоковое описание
 sensitive(b); //Функциональное описание
 sensitive(e); //Функциональное описание
 }
 };
```

Примечание: Во избежание риска возникновения ошибок на этапе симуляции проекта, включайте все входные порты в список чувствительных портов при реализации комбинаторной логики. В примере показана ситуация с неполным перечнем входов, отображенном в списке чувствительных портов.

```
void comb_proc ()
{
```

```
out_x = in_a & in_b & in_c;
}
SC_CTOR( comb_logic_complete )
{
SC_METHOD( comb_proc);
sensitive << in_a << in_b; // пропущен in_c
}
```

Пример описания заголовочного файла модели элемента 3И на SystemC:

```
#ifndef AND3_H
#define AND3_H
#include "systemc.h"

SC_MODULE(and3)
{
sc_in <bool> A, B, C;
sc_out <bool> F;
void do_and3()
{
F.write( A.read() && B.read() && C.read() );
}
SC_CTOR(and3)
{
SC_METHOD(do_and3);
sensitive << A << B << C;
}
};
#endif
```

Конструкция `Edge-Sensitive` используется для реализации последовательной логики, при моделировании триггеров. Для этого необходимо использовать такие потоки как `sensitive_neg (срез)`, `sensitive_pos (фронт)`. Входные порты должны иметь тип `sc_in<bool>`, ниже приведен пример использования конструкции `edge-sensitive`:

```
SC_MODULE(my_module)
{
 sc_in<int> a;
 sc_in<bool> b;
 sc_in<bool> clock;
 sc_out<int> y;
 sc_in<bool> reset;
 sc_signal<bool>c;
```

```
 void my_method_proc();
 SC_CTOR(my_module)
{
 SC_METHOD(my_method_proc);
 sensitive_pos (clock); /* Функциональное
описание*/
 sensitive_neg << b << reset; /* Потоковое
описание*/
}
};
```

Ограничения при использовании Sensitivity Lists:

- Нельзя совмещать конструкции Level-Sensitive и Edge-Sensitive в одном процессе;
- Нельзя применять тип sc_logic для реализации синхросигнала (clock) или других Edge-Sensitive’s входов. Допустимым является только тип sc_in<bool>.

Листинг 3.1

Пример описания заголовочного файла JK-триггера на SystemC

```
#ifndef jk_H
#define jk_H

#include "systemc.h"
SC_MODULE(jk)
{
sc_in <bool> J, K, Clock, Reset;
 sc_out <bool> F, NF;
void do_jk()
{
if(!Clock.read())
{
if(J.read() == true && K.read() == true)
{ if(F.read())
{ F.write(false);
NF.write(true);
}
else
{
F.write(true);
NF.write(false);
}
```

```
}
if(J.read() == true && K.read() == false)
 {
F.write(true);
NF.write(false);
}
if(J.read() == false && K.read() == true)
 {
F.write(false); NF.write(true);
 }
 }
if(!Reset.read())
{
F.write(false);
NF.write(true);
}
}
SC_CTOR(jk)
{ SC_METHOD(do_jk);
sensitive << Clock.neg();
sensitive << Reset.neg();
}
};
#endif
```

Определения списка чувствительности процесса:

- чувствительный к оператору ();
  - принимает один порт или сигнал в качестве аргумента;
  - sensitive (sig1); sensitive (sig2); sensitive (sig3);
- чувствительный к обозначению потока;
  - принимает произвольное количество аргументов;
  - sensitive << sig1 << sig2 << sig3;
- sensitive_pos с помощью () или << operator
  - определяет чувствительность к положительному фронту булевого сигнала или такта:
    sensitive_pos << clk;
- sensitive_neg с () или << оператором;
  - определяет чувствительность к отрицательному фронту булевого сигнала или такта sensitive_neg << clk;

3.9.1. Процесс SC_THREAD

Самый простой тип процесса это поток SystemC, названный SC_THREAD. Концептуально поток SystemC идентичен потоку программного обеспечения. В простых C/C++ программах есть только один поток, работающий для всей программы. Ядро SystemC позволяет выполнять много потоков параллельно (это называется параллелизм) Простой SC_THREAD начинает выполняться, когда планировщик называет его, и заканчивается, когда происходит выход или возврат. SC_THREAD вызывается только один раз, так же, как простая программа C/C++. SC_THREAD может также приостановить себя.

Регистрация простого процесса: SC_THREAD

После того, как вы определили тип процесса, необходимо определить и зарегистрировать его с ядром моделирования. Этот шаг позволяет планировщику симуляции ядра вызывать поток. Регистрация происходит в модуле конструктора класса SC_CTOR. Регистрация потока SystemC кодируется с помощью макроса C++ SC_THREAD внутри конструктора следующим образом:

```
SC_THREAD(process_name);/*Должно быть внутри конструктора*/
```

`Process_name` - это имя соответствующего метода члена класса.

C++ позволяет конструктору появляются до или после провозглашения процесса метода. Ниже приведен полный пример определения SC_THREAD в модуле:

```
//FILE: simple_process_ex.h
SC_MODULE(simple_process_ex) {
SC_CTOR(simple_process_ex) {
SC_THREAD(my_thread_process);
}
void my_thread_process(void);
};
```

Member function – функция- член: функция, которая является элементом класса и которая оперирует с объектами этого класса, адресуясь через указатель this.

Изучать методы и процессы мы будем, используя в примерах программы для простого счетчика **first_counter**, детально описанные на сайте компании ASIC (http://www.asic-world.com/systemc/index.html). Автором обучающих программ является Deepak Kumar Tala, любезно разрешивший свободно их использовать при условии указания ссылки на его сайт www.asic-world.com. Практически все программы, представленные в этой главе листингами, были заимствованы нами с сайта www.asic-world.com. Мы исправили некоторые ошибки, отладили

программы, получили правильные решения, сдалали перевод комментариев и представляем Вам эти программы для изучения.

Схема счетчика показана на рис. 3.4. Управляя входными сигналами clock, reset, enable и условиями реакции счетчика на эти сигналы, можно проектировать различные варианты процессов и моделированием проверять правильность их работы. Управляющие сигналы создает программа Testbench, которая служит главной программой `main.cpp.`

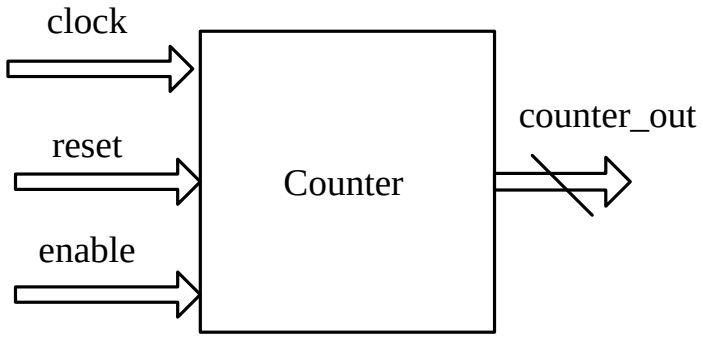


Рис. 3.4. Первый счетчик-first counter

Листинг 3.2

Программа sc_counter_threads.cpp

```
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in_clk     clock ;  /* Входной тактовый
сигнал проекта*/
      sc_in<bool>   reset ;    /* активный высокий,
синхронный вход сброса */
      sc_in<bool>   enable;      /* Активный высокий
разрешающий сигнал для счетчика */
      sc_out<sc_uint<4> > counter_out; /* 4-битный
векторный выход счетчика */

      //--------Локальные переменные здесь ----------
-----------
      sc_uint<4> count;

      //--------Код начинается здесь ----------
      /* Ниже функция реализует фактическую логику
счетчика*/
      void incr_count () {
```

```
        /* Для потоков мы должны иметь цикл во время
«true-истина»*/
        while (true) {
          /* Ожидание пока произойдет событие в
списке чувствительности*/
          /* В этом примере - положительный фронт
тактов*/
          wait();
          if (reset.read() == 1) {
            count =  0;
            counter_out.write(count);
          /* Если включена функция enable, то мы
увеличиваем счетчик */
          } else if (enable.read() == 1) {
            count = count + 1;
            counter_out.write(count);
          }
        }
      } // Конец функции incr_count

      /* Ниже функции печатают значение count, когда
оно изменяется */
      void print_count () {
        while (true) {
          wait();
          cout<<"@" << sc_time_stamp() <<
            " :: Counter Value
"<<counter_out.read()<<endl;
        }
      }

      // Конструктор для счетчика
      /* Поскольку этот счетчик переключается одним
положительным фронтом, мы запускаем нижний блок
относительно положительного фронта такта*/
      SC_CTOR(first_counter) {
        // Чувствительность к фронту такта
        SC_THREAD(incr_count);
        sensitive << clock.pos();
        /* Чувствительность к изменению уровня
выходного сигнала счетчика */
SC_THREAD(print_count);
```

```
        sensitive << counter_out;
      } // Конец конструктора

    }; // Конец модуля счетчика

    // Испытательный стенд для sc_counter_threads

    int sc_main (int argc, char* argv[]) {
      sc_signal<bool>   clock;
      sc_signal<bool>   reset;
      sc_signal<bool>   enable;
      sc_signal<sc_uint<4> > counter_out;
      int i = 0;
      // Подключить DUT
      first_counter counter("COUNTER");
        counter.clock(clock);
        counter.reset(reset);
        counter.enable(enable);
        counter.counter_out(counter_out);
          sc_start(1, SC_MS);

      // Инициализировать все переменные
      reset = 0;       // начальное значение сброса
      enable = 0;      /* начальное значение
разрешения*/
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 1;    // Установить сброс
      cout << "@" << sc_time_stamp() <<" Asserting
reset\n" << endl;
      for (i=0;i<10;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 0;    // Отменить сброс
```

```
      cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" Asserting
Enable\n" << endl;
      enable = 1;  // Разрешение активации
      for (i=0;i<20;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
      enable = 0; // Разрешить запрет
        cout << "@" << sc_time_stamp() <<"
Terminating simulation\n" << endl;
      return 0;// Завершить симуляцию

     }
```

В модели счетчик выполняет счет при отсутствии сигнала reset и наличии сигнала enable. Программа testbench создает входные сигналы, выводит выходное значение со счетчика, регистрирует текущее время. Результаты моделирования, полученные нами в среде Eclipse, показаны на рис. 3.5

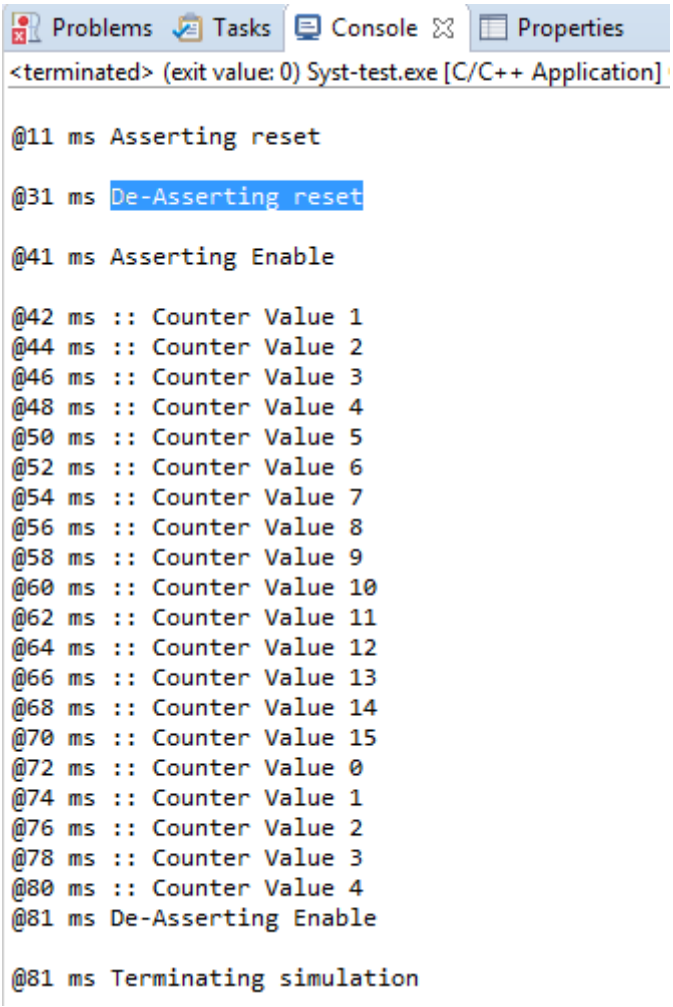


Рис. 3.5. Результаты моделтрования в среде Eclipse

3.9.2. Процесс SC_METHOD

Процесс SC_METHOD в некотором смысле проще, чем SC_THREAD. Однако, эта простота делает его использование более трудным для некоторых стилей моделирования. По возможностям SC_METHOD является более эффективным, чем SC_THREAD. Одним из основных отличий является вызов. Процессы SC_METHOD никогда не приостанавливаются внутренне (то есть, они никогда не могут вызывать ожидание `wait()`). Вместо этого SC_METHOD процессы выполняются полностью с возвратом. Симулятор вызывает их неоднократно на основе динамической или статической чувствительности.

Поскольку процессам SC_METHOD запрещаются приостановления изнутри, они не могут вызвать метод ожидания. Попытка вызова ожидания либо непосредственно, либо из результатов SC_METHOD приводит к ошибкам во время выполнения. Они известны как блокирование методов. Методы чтения и записи данных типа `sc_fifo`, которые обсуждается

далее, являются примерами способов блокировки. Таким образом, SC_METHOD процессы должны избегать использования вызовов методов блокирования. Синтаксис SC_METHOD процессов практически идентичен SC_THREAD за исключением ключевого слова SC_METHOD:

```
SC_METHOD(process_name);/*Расположен внутри конструктора*/
```

Методы ведут себя как функции. Когда функция вызывается, она запускается, выполняется и возвращает исполнение обратно в механизм вызова. Метод вызывается, когда изменяется какое-либо событие в списке чувствительности. Запуск события в чувствительном списке может быть либо чувствительным к краю (фронту), либо чувствительным к уровню.

Примечание: сигналы чувствительных списков, которые запускают процесс, могут быть сигналами, или локальной переменной, или портом.

Входные сигналы, вызывающие повторную активацию процесса, задаются списком чувствительности. Список чувствительности указан в конструкторе модуля

Листинг 3.3

Программа sc_counter_method.cpp

```
#include "systemc.h"

int sc_main (int argc, char* argv[]) {
SC_MODULE (first_counter) {
  sc_in_clk     clock ;      /*  Входной сигнал тактов проекта*/
  sc_in<bool>   reset ;      /* активный высокий, синхронный вход сброса */
  sc_in<bool>   enable;      /* Активный высокий разрешающий сигнал для счетчика */
  sc_out<sc_uint<4> > counter_out; /* 4-битный векторный выход счетчика */

  //------Локальные переменные здесь -----
  sc_uint<4> count;

  //------Код начинается здесь --------
  /* Ниже функция реализует фактическую логику счетчика*/
  void incr_count () {
    /* На каждом нарастающем фронте тактов мы проверяем, активен ли сброс */
```

```
        /* Если активен, мы загружаем в счетчик
4'b0000*/
        if (reset.read() == 1) {
          count =  0;
          counter_out.write(count);
        /* Если включена функция enable, то мы
увеличиваем счетчик */
        } else if (enable.read() == 1) {
          count = count + 1;
          counter_out.write(count);
        }
      } // Конец функции incr_count

      /* Ниже функция печатает значение count, когда
оно изменяется */
      void print_count () {
        cout<<"@" << sc_time_stamp() <<
          " :: Counter Value
"<<counter_out.read()<<endl;
      }

      // Конструктор для счетчика
      /*Поскольку этот счетчик запускается
положительным фронтом и срабатывает один раз, мы
запускаем нижний блок с учетом положительного фронта
тактов, а также когда сброс изменяет состояние*/
      SC_CTOR(first_counter) {
        // Чувствительность по фронту и уровню
        SC_METHOD(incr_count);
        sensitive << reset;
        sensitive << clock.pos();
        // Метод, чувствительный к уровню
        SC_METHOD(print_count);
        sensitive << counter_out;
      } // Конец конструктора

    }; // Конец модуля счетчика
    };

    //Испытательный стенд для sc_counter_method
    int sc_main (int argc, char* argv[]); {
      sc_signal<bool>   clock;
```

```
      sc_signal<bool>   reset;
      sc_signal<bool>   enable;
      sc_signal<sc_uint<4> > counter_out;
      int i = 0;
      // Подключить DUT
      first_counter counter("COUNTER");
        counter.clock(clock);
        counter.reset(reset);
        counter.enable(enable);
        counter.counter_out(counter_out);
          sc_start(1, SC_MS);


      // Инициализировать все переменные
      reset = 0;       // начальное значение сброса
      enable = 0;      /* начальное значение
разрешения*/

      reset = 1;    // Установить сброс
      cout << "@" << sc_time_stamp() <<" Asserting
reset\n" << endl;
      for (i=0;i<3;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 0;    // Отменить сброс
      cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
      for (i=0;i<4;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" Asserting
Enable\n" << endl;
      enable = 1;  // Разрешить активацию
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
```

```
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
      enable = 0; // Разрешить запрет
        cout << "@" << sc_time_stamp() <<"
Terminating simulation\n" << endl;
        return 0;// Завершить симуляцию

    };
    };
```

Результаты моделирования:

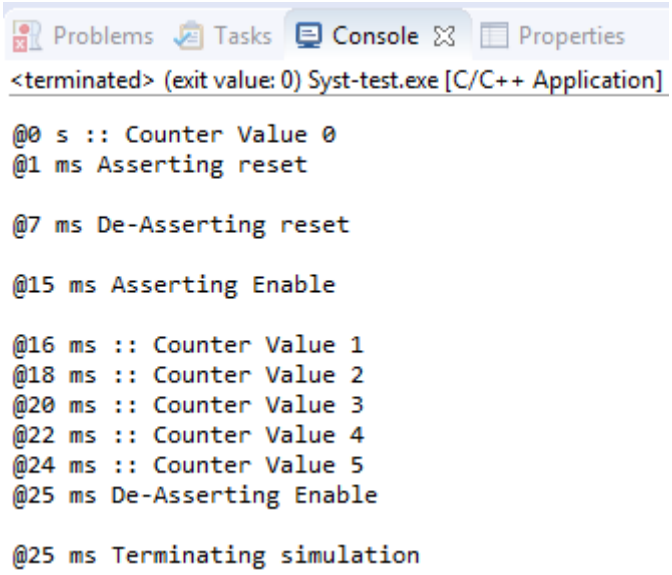


Рис. 3.6. Результаты моделирования в Eclipse

Рекомендуем сверить текст программы с результатом моделирования и убедиться в соответствии результатов коду программы.

3.9.3. Процесс SC_CTHREAD

Вариацией процесса SC_THREAD с тактовой частотой является популярный для поведенческих инструментов синтеза SC_CTHREAD,. Эта популярность отчасти потому, что синтезированные логические инструменты в настоящее время производят для полностью синхронного кода, и SC_CTHREAD предоставляет некоторые новые возможности для упрощения кодирования.

```
SC_CTOR(module_name) {
SC_CTHREAD(NAME_cthread, clock_name.edge()) ;
}
```

Процесс SC_CTHREAD отличается от процесса SC_THREAD несколькими способами. Сначала процесс SC_CTHREAD указывает объект тактирования. Когда другие типы процессов описывают в конструкторе модуля, они имеют только имя указанного процесса, а процесс SC_CTHREAD имеет имя процесса и такты, которые запускают процесс. У SC_CTHREAD нет отдельного списка чувствительности, как у других типов процессов. Список чувствительности - это только заданный фронт тактового сигнала. Процесс SC_CTHREAD будет активизироваться всякий раз, когда происходит указанный фронт тактового сигнала. В этом примере указан положительный фронт такта, поэтому процесс `incr_count` будет выполняться на каждом положительном фронте тактового сигнала.

Примечание: SC_CTHREAD может иметь только три бита портов как триггер.

Пример программы с процессом SC_CTHREAD приведен в листинге 3.4

Листинг 3.4

Программа sc_counter_cthread.cpp

```
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in_clk     clock ;   /* Входной тактовый
сигнал проекта */
  sc_in<bool>   reset ;      /* активный высокий,
синхронный вход сброса */
  sc_in<bool>   enable;      /* Активный высокий
разрешающий сигнал для счетчика */
  sc_out<sc_uint<4> > counter_out; /* 4-битный
векторный выход счетчика */

  //--- Локальные переменные здесь -----
  sc_uint<4> count;

  //---------- Код начинается здесь ---------
  /* Ниже функция реализует фактическую логику
счетчика*/
  void incr_count () {
```

```
        // Для потоков мы должны иметь цикл во время
«true» цикл
        while (true) {
          /* Подождите, пока произойдет событие в
списке чувствительности */
      // В этом примере - положительный фронт такта
          wait();
          if (reset.read() == 1) {
            count =  0;
            counter_out.write(count);
          /* Если включена функция enable, то мы
увеличиваем счетчик */
          } else if (enable.read() == 1) {
            count = count + 1;
            counter_out.write(count);
          }
        }
      } // Конец функции incr_count

      /* Ниже функция печатает значение count, когда
оно изменяется */
      void print_count () {
        while (true) {
          wait();
          cout<<"@" << sc_time_stamp() <<
            " :: Counter Value
"<<counter_out.read()<<endl;
        }
      }

      // Конструктор для счетчика
      /*Поскольку этот счетчик запускается
положительным фронтом один раз, мы запускаем нижний
блок по положительному фронту такта* /
        SC_CTOR(first_counter) {
        /* cthreads требуют наличия имени потока и
запуска */
        // событие, переданное как объект тактов
        SC_CTHREAD(incr_count, clock.pos());
        /* Чувствительность по уровню к изменению
выходного сигнала счетчика*/
        SC_THREAD(print_count);
```

```
        sensitive << counter_out;
    } // Конец конструктора

}; // Конец модуля counter


//Испытательный стенд для sc_counter_cthread.cpp
int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Подключить DUT
  first_counter counter("COUNTER");
    counter.clock(clock);
    counter.reset(reset);
    counter.enable(enable);
    counter.counter_out(counter_out);
      sc_start(1, SC_MS);


  // Инициализировать все переменные
  reset = 0;       // начальное значение сброса
  enable = 0;      /* начальное значение
разрешения*/
  for (i=0;i<2;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  reset = 1;    // Установить сброс
  cout << "@" << sc_time_stamp() <<" Asserting
reset\n" << endl;
  for (i=0;i<2;i++) {
    clock = 0;
    sc_start(1, SC_MS);
    clock = 1;
    sc_start(1, SC_MS);
  }
  reset = 0;    // Отменить сброс
```

```
      cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
      for (i=0;i<3;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" Asserting
Enable\n" << endl;
      enable = 1;  // Разрешение активации
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
      enable = 0; // Разрешить запрет
        cout << "@" << sc_time_stamp() <<"
Terminating simulation\n" << endl;

      return 0;// Завершить симуляцию

    }
```

Результаты моделирования;

```
@5 ms Asserting reset

@9 ms De-Asserting reset

@15 ms Asserting Enable

@16 ms :: Counter Value 1
@18 ms :: Counter Value 2
@20 ms :: Counter Value 3
@22 ms :: Counter Value 4
@24 ms :: Counter Value 5
@25 ms De-Asserting Enable

@25 ms Terminating simulation
```

Рис. 3.7. Результаты моделирования в среде Eclipse

Процесс и потоки (не cthreads) автоматически запускаются в конструкторе, даже если событие в списке чувствительности не возникает. Чтобы предотвратить это неумышленное выполнение, используйте функцию `dont_initialize ()`, как показано в примере.

Листинг 3.5

Программа sc_dont_initialize.cpp

```
#include "systemc.h"

SC_MODULE (tff_sync_reset) {
  sc_in    <bool> data, clk, reset  ;
  sc_out   <bool> q;

  bool q_l ;

  void tff () {
    if (reset.read()) {
      q_l = 0;
    } else if (data.read()) {
      q_l = !q_l;
    }
    q.write(q_l);
  }

  SC_CTOR(tff_sync_reset) {
    SC_METHOD (tff);
      dont_initialize();
      sensitive << clk.pos();
  }

};

SC_MODULE (tff_tb) {
  sc_in<bool> clk;

  sc_signal <bool> data, reset  ;
  sc_signal   <bool> q;
  tff_sync_reset *dut;

  void do_test() {
    cout << "@" << sc_time_stamp() <<" Starting
test"<<endl;
```

```
        wait();
        cout << "@" << sc_time_stamp() <<" Asserting
reset"<<endl;
        reset = true;
        wait(4);
        cout << "@" << sc_time_stamp() <<" De-
Asserting reset"<<endl;
        reset = false;
        wait(3);
        cout << "@" << sc_time_stamp() <<" Asserting
Data input"<<endl;
        data = true;
        wait(3);
        data = false;
        cout << "@" << sc_time_stamp() <<" De-
Asserting Data input"<<endl;
        wait(3);
        cout << "@" << sc_time_stamp() <<"
Terminating simulation"<<endl;
        sc_stop();
      }

      SC_CTOR(tff_tb) {
        dut = new tff_sync_reset ("TFF");
        dut->clk    (clk);
        dut->reset  (reset);
        dut->data   (data);
        dut->q      (q);
        SC_THREAD (do_test);
          dont_initialize();
          sensitive << clk.pos();
      }
    };

    int sc_main (int argc, char* argv[]) {
      sc_clock clock ("my_clock",1,0.5);

      tff_tb  object("TFF_TB");
        object.clk (clock);
      sc_trace_file *wf =
sc_create_vcd_trace_file("tff");
        sc_trace(wf, object.clk, "clk");
```

```
    sc_trace(wf, object.reset, "reset");
    sc_trace(wf, object.data, "data");
    sc_trace(wf, object.q, "q");

  sc_start(0);
  sc_start();
  sc_close_vcd_trace_file(wf);
  return 0;// Завершить симуляцию
}
```

Результаты моделирования:

```
Info: (I702) default timescale unit used for tracing: 1 ps (tff.vcd)
@0 s Starting test
@1 ns Asserting reset
@5 ns De-Asserting reset
@8 ns Asserting Data input
@11 ns De-Asserting Data input
@14 ns Terminating simulation

Info: /OSCI/SystemC: Simulation stopped by user.
```

Рис. 3.8. Результаты моделирования

Некоторыми из более простых объектов, предоставляемых этим новым процессом, являются новая форма ожидания `wait(N)` и уровня чувствительных ожиданий, которая называется `wait_until ()`. Синтаксисы выглядят так:

```
wait(N); // задержка N фронтов тактов
wait_until(delay_expr); // пока expr true @ clock
```

3.10. Глобальное и локальное наблюдение

Наибольший интерес в том, что SC_CTHREAD обеспечивает концепцию наблюдения сигналов, которые эффективно изменяют поведение от `wait().` Когда наблюдают активный сигнал, выполнение переходит к новой области после возвращения из ожидания `wait ()`, а не переходит к следующей команде. SystemC имеет две формы наблюдения: глобальную и локальную. Самая простая форма наблюдения носит глобальный характер. Глобальное наблюдение следит за активностью сигнала, чтобы перезапустить тактируемый поток с самом начала.

Более управляемые формы наблюдения предполагает использование четырех макросов: `W_BEGIN`, `W_DO`, `W_ESCAPE` и `W_END`. Эти макросы ранее использовались в качестве набора в указанном порядке, и они определяли три области в коде наблюдения.

В первой области сигналы для слежения объявляются. В второй области, код слежения кодируется. В последнем области обеспечивается обрабатка наблюдаемой активации сигнала. Вот краткий обзор синтаксиса:

```
W_BEGIN
watching(delay_expr);
W_DO
Code_being_watched
W_ESCAPE
Code_handling_escape_condition
W_END
```

В наших программах, основванных на библиотеке SystemC-2.3.1, эти макросы отсутствуют.

Синтаксис `wait ()` и `wait_until ()` требует выражения задержки, `delay_expr`, который должен быть выражен с использованием сигналов с задержкой. В версиях SystemC перед стандартизацией существует еще один синтаксис для `wait (N)` и чувствительный к уровню ожидания, называемый `wait_until ()`. Вы можете увидить это в устаревшем коде.

Это метод `delayed ()` – специальный метод, который дает значение в конце дельта-цикла. Имейте в виду, что этот метод устарел, отсутствует в новом стандарте, и этот синтаксис применим только к версиям до версии OSCI 2.2. Метод `delayed ()`, можно заменить почти эквивалентным следующим кодом SC_THREAD, предполагая, что поток статически чувствительный к краю тактового сигнала:

```
for(i=0;i!=N;i++) wait();//аналогично wait (N)
do wait() while(!expr);/*также как
wait_until(dexpr)*/
```

Листинг 3.6

Программа sc_wait_until

```
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in<bool>   clock ;      /* Входной сигнал
тактов проекта*/
  sc_in<bool>   reset ;      /* активный высокий,
синхронный вход сброса*/
  sc_in<bool>   enable;      /* Активный высокий
разрешающий сигнал для счетчика*/
  sc_out<sc_uint<4> > counter_out; /* 4-битный
векторный выход счетчика*/
```

```
      //----- Локальные переменные здесь ------
      sc_uint<4> count;

      //----- Код начинается здесь ---------
      /* Ниже функция реализует фактическую логику
счетчика*/


      void incr_count () {
        /* Для потоков мы должны иметь оператор while
(true) для  цикла */
        while (true) {
          /* Подождите, пока остался true или enable
остается  true*/
          /* Каждый wait_until задерживает выполнение
одним clock.pos ()*/
        do wait();
        while(reset == false||enable==true);/* также
как wait_until(dexpr)*/


    if (reset.read() == 1) {

    count = count + 1;
          counter_out.write(count);
          /* Если включена функция enable, то мы
увеличиваем счетчик */
          } else if (enable.read() == 1) {
            count = count + 1;
            counter_out.write(count);
          }
        }
      } // Конец функции incr_count

      /* Ниже функции печатают значение count, когда
оно изменяется */
      void print_count () {
        while (true) {
          wait();
          cout<<"@" << sc_time_stamp() <<
```

```
            " :: Counter Value
"<<counter_out.read()<<endl;
        }
      }

      // Конструктор для счетчика
      /*Поскольку этот счетчик переключается
положительным фронтом, мы запускаем нижний блок по
положительному фронту такта*/
      SC_CTOR(first_counter) {
        // Чувствительность к фронту такта
        SC_CTHREAD(incr_count, clock.pos());
        sensitive <<clock.pos();
        /* Чувствительность по уровню к изменению
выходного сигнала счетчика*/
        SC_THREAD(print_count);
        sensitive << clock.pos();
      } // Конец конструктора

    }; // Конец модуля счетчика


    //Испытательный стенд для sc_wait_until

    int sc_main (argc, char* argv[]) {
      sc_signal<bool>   clock;
      sc_signal<bool>   reset;
      sc_signal<bool>   enable;
      sc_signal<sc_uint<4> > counter_out;
      int i = 0;
      // Подключить  DUT
      first_counter counter("COUNTER");
        counter.clock(clock);
        counter.reset(reset);
        counter.enable(enable);
        counter.counter_out(counter_out);
          sc_start(1, SC_MS);

      // Инициализировать все переменные
      reset = 1;   // Начальное значение сброса
      enable = 1;  // Начальное значение разрешения
```

```
      cout << "@" << sc_time_stamp() <<" Asserting
reset_enable\n" << endl;
      for (i=0;i<2;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 1;  // Установить сброс
      enable = 0;
      cout << "@" << sc_time_stamp() <<"De- Asserting
enable\n" << endl;
      for (i=0;i<3;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 0;    // Отменить сброс
      enable = 0;
      cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }

      reset = 0;
      enable = 1;  // Установить разрешение
      cout << "@" << sc_time_stamp() <<" Asserting
Enable\n" << endl;
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
       cout << "@" << sc_time_stamp() <<" Terminating
simulation\n" << endl;
```

```
    return 0;// Завершить симуляцию
}
```

```
@1 ms Asserting reset_enable

@2 ms :: Counter Value 0
@4 ms :: Counter Value 0
@5 ms  De- Asserting enable

@6 ms :: Counter Value 0
@8 ms :: Counter Value 1
@10 ms :: Counter Value 2
@11 ms De-Asserting reset

@12 ms :: Counter Value 3
@14 ms :: Counter Value 3
@16 ms :: Counter Value 3
@18 ms :: Counter Value 3
@20 ms :: Counter Value 3
@21 ms Asserting Enable

@22 ms :: Counter Value 3
@24 ms :: Counter Value 3
@26 ms :: Counter Value 3
@28 ms :: Counter Value 3
@30 ms :: Counter Value 3
@31 ms Terminating simulation
```

Рис. 3.9. Результаты моделирования

Глобальное наблюдение

Процессы SC_CTHREAD обычно имеют бесконечные циклы, которые будут выполняться непрерывно. Дизайнер обычно хочет каким-то образом инициализировать поведение цикла или выйти из цикла при возникновении условия. Это достигается с помощью наблюдательной конструкции. Наблюдающая конструкция будет контролировать указанное условие. Когда возникает это условие, управление переносится из текущей точки выполнения в начало процесса, где можно наблюдать появление наблюдаемого условия. Наблюдающая конструкция доступна только для процессов SC_CTHREAD.

Одним из неожиданных последствий выхода из цикла `while` и начала процесса является то, что все переменные, локально определенные внутри процесса, потеряют свое значение. Если значение переменной необходимо сохранить между вызовами процесса, объявите переменную в модуле процесса, а не локальную для процесса.

Пример глобального просмотра

В конструкторе примера приведен следующий оператор: watching(reset == true). Этот оператор указывает, что для этого процесса

будет наблюдаться сброс сигнала. Если сброс сигнала изменится на `true`, наблюдающее выражение будет истинным, и планировщик SystemC остановит выполнение цикла `while` для этого процесса и начнет выполнение в первой строке процесса.

Листинг 3.7

Программа sc_global_watching

```
#include "systemc.h"

SC_MODULE (first_counter) {
  sc_in_clk     clock ;      /* Входной тактовый
сигнал проекта*/
  sc_in<bool>   reset ;      /* Активный высокий,
синхронный вход сброса */
  sc_in<bool>   enable;      /* Активный высокий
разрешающий сигнал для счетчика */
  sc_out<sc_uint<4> > counter_out; /* 4-битный
векторный выход счетчика */

  //-----Локальные переменные здесь----
  sc_uint<4>   count;

  //-----Код начинается здесь-----------
  /* Ниже функция реализует фактическую логику
счетчика*/
  void incr_count () {
    if (reset.read() == 1) {
      count =  0;
      counter_out.write(count);
      cout<<"@" << sc_time_stamp() <<
        " :: Watching reset is activated "<<endl;
    }
    while (true) {
      wait();
      /* Если включена функция enable, то мы
увеличиваем счетчик */
      if (enable.read() == 1) {
        count = count + 1;
        counter_out.write(count);
      }
    }
  } // Конец функции incr_count
```

```
      /* Ниже функции печатает значение count, когда
оно изменяется */
      void print_count () {
        while (true) {
          wait();
          cout<<"@" << sc_time_stamp() <<
            " :: Counter Value
"<<counter_out.read()<<endl;
        }
      }

      // Конструктор для счетчика
      /*Поскольку этот счетчик переключается
положительным фронтом, мы запускаем нижний блок по
положительному фронту такта*/
      SC_CTOR(first_counter) {
        /* cthreads требуют наличия имени потока и
запуска */
        // Событие, переданное как объект такта
        SC_CTHREAD(incr_count, clock.pos());
          watching(reset == true);
        /* Чувствительность по уровеню к изменению
выходного сигнала счетчика*/
        SC_THREAD(print_count);
          sensitive << counter_out;
      } // Конец конструктора

    }; // Конец модуля счетчика
```

Локальное наблюдение

Локальное наблюдение позволяет вам точно указать, какая часть процесса наблюдает какие-либо сигналы и где расположены обработчики событий. Эта функциональность задается 4 макросами, которые определяют границы каждой из областей. Ниже приведен псевдосинтаксис локального просмотра.

```
    W_BEGIN
      // разместите здесь показания просмотра
      watching(...);
      watching(...);
    W_DO
```

```
      // Здесь происходит функциональность процесса
      ...
    W_ESCAPE
      /* Здесь отправляются обработчики для
отслеживаемых событий*/
      if (..) {
        ...
      }
    W_END
```

Макрос W_BEGIN отмечает начало локального блока наблюдения. Между макросами W_BEGIN и W_DO находятся все объявления просмотра. Эти объявления выглядят так же, как глобальные события просмотра. Между макросом W_DO и макросом W_ESCAPE находится место, где размещаются функциональные возможности процесса. Это код, который выполняется, пока не происходит ни одно из событий наблюдения. Между макросами W_ESCAPE и W_END находятся обработчики событий. Обработчики событий проведут проверку, чтобы убедиться, что произошло соответствующее событие, а затем выполните необходимые действия для этого события. Макрос W_END завершает локальный блок наблюдения.

Ранее мы отмечали, что указанные макросы отсутствуют в нашей версии SystemC-2.3.1.

Есть несколько интересных вещей, которые возможны при локальном наблюдении:

Все события в блоке объявления имеют одинаковый приоритет. Если нужен другой приоритет, тогда локальные блоки просмотра должны быть вложены.

Локальный просмотр работает только в процессах SC_CTHREAD.

Сигналы в наблюдающих выражениях отбираются только по активным границам процесса. В процессе SC_CTHREAD это означает, что процесс чувствителен к изменениям только в тактах.

Наблюдаемые в глобальном масштабе события имеют более высокий приоритет, чем локально наблюдаемые события.

Без макросов W_BEGIN, W_DO, W_ESCAPE, W_END не работает оператор watching (). Поэтому на листинге 3.8 приведена отредактированная нами программа, в которой исключены неработающие операторы. Программа использует условные операторы и дает те же результаты моделирования.

Листинг 3.8

Программа sc_local_watching.cpp

```
    include "systemc.h"

    SC_MODULE (first_counter) {
      sc_in<bool>   clock ;      /* Входной nfrnjdsq
сигнал проекта */
      sc_in<bool>   reset ;      /* Активный высокий,
синхронный вход сброса */
      sc_in<bool>   enable;      /* Активный высокий
разрешающий сигнал для счетчика */
      sc_out<sc_uint<4> > counter_out; /* 4-битный
векторный выход счетчика */

      //-------Локальные переменные здесь ------
      sc_uint<4> count;

      //-----Код начинается здесь ---------------
      /* Ниже функция реализует фактическую логику
счетчика */
      void incr_count () {
        while (true) {
          wait();
          /* Если включена функция enable, то мы
увеличиваем счетчик */
          if (enable.read() == 1) {
            count = count + 1;
            counter_out.write(count);
          }

            wait(2, SC_MS);
            if (enable.read() == 1) {
              count = count + 1;
              counter_out.write(count);
            }

            if (reset.read() == 1) {
              count =  0;
              counter_out.write(count);
              cout<<"@" << sc_time_stamp() <<
                " :: Local Watching reset is
activated"<<endl;
            }
```

```
        }
      } // Конец функции incr_count

      /* Ниже функции печатает значение count, когда
оно изменяется */
    void print_count () {
        while (true) {
          wait();
          cout<<"@" << sc_time_stamp() <<
            " :: Counter Value
"<<counter_out.read()<<endl;
        }
      }

      // Конструктор для счетчика
    /*Поскольку этот счетчик запускается
положительным фронтом, связанным с ним,мы запускаем
нижний блок по положительному фронту тактов */
      SC_CTOR(first_counter) {
        /* cthreads требуют, чтобы имя потока и
инициирующее событие передавались как объект
синхронизации */
        SC_CTHREAD(incr_count, clock.pos());
      /* Чувствительность по уровню к изменению
выходного сигнала счетчика*/
        SC_THREAD(print_count);
          sensitive << counter_out;
      } // Конец конструктора

    }; // Конец модуля счетчика

    // Испытательный стенд
    int sc_main (int argc, char* argv[]) {
      sc_signal<bool>   clock;
      sc_signal<bool>   reset;
      sc_signal<bool>   enable;
      sc_signal<sc_uint<4> > counter_out;
      int i = 0;
      // Подключить DUT
      first_counter counter("COUNTER");
        counter.clock(clock);
```

```
        counter.reset(reset);
        counter.enable(enable);
        counter.counter_out(counter_out);
          sc_start(1, SC_MS);


      // Initialize all variables
      reset = 0;    // начальное значение сброса
      enable = 0;   // начальное значение разрешения
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 1;    // Установить сброс
      cout << "@" << sc_time_stamp() <<" Asserting
reset\n" << endl;
      for (i=0;i<10;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 0;    // Отменить сброс
      cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" Asserting
Enable\n" << endl;
      enable = 1;  // Установить разрешение
      for (i=0;i<20;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
```

```
      cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
      enable = 0; // Отменить разрешение
        cout << "@" << sc_time_stamp() <<"
Terminating simulation\n" << endl;

      return 0;// Завершить симуляцию

}
```

```
@11 ms Asserting reset

@14 ms :: Local Watching reset is activated
@18 ms :: Local Watching reset is activated
@22 ms :: Local Watching reset is activated
@26 ms :: Local Watching reset is activated
@30 ms :: Local Watching reset is activated
@31 ms De-Asserting reset

@41 ms Asserting Enable

@42 ms :: Counter Value 1
@44 ms :: Counter Value 2
@46 ms :: Counter Value 3
@48 ms :: Counter Value 4
@50 ms :: Counter Value 5
@52 ms :: Counter Value 6
@54 ms :: Counter Value 7
@56 ms :: Counter Value 8
@58 ms :: Counter Value 9
@60 ms :: Counter Value 10
@62 ms :: Counter Value 11
@64 ms :: Counter Value 12
@66 ms :: Counter Value 13
@68 ms :: Counter Value 14
@70 ms :: Counter Value 15
@72 ms :: Counter Value 0
@74 ms :: Counter Value 1
@76 ms :: Counter Value 2
@78 ms :: Counter Value 3
@80 ms :: Counter Value 4
@81 ms De-Asserting Enable

@81 ms Terminating simulation
```

Рис. 3.10. Результаты моделирования sc_local_watching

Процесс может быть либо сделан чувствительным к уровню логики, либо может быть сделать чувствительным к фронту.

Процессы можно заставить работать вечно. Такие процессы мы называем потоками в SystemC. Можно запускать, когда происходит какое-либо событие (пример: `posedge of clock`).

Примечание: сигналы списка чувствительности, которые запускают процесс / поток, должны быть портами и не могут быть сигналом или локальной переменной.

3.11. Порты и сигналы

Порты модуля являются внешним интерфейсом, который передает информацию из модуль и в модуль и переключает действия внутри модуля. Сигналы создают связи между портами модулей, что позволяет модулям взаимодействовать.

Порт может иметь три различных режима работы:

• Ввод
• Вывод
• Ввод-Вывод.

Входной порт (In) передает данные в модуль. Выходной порт (Out) передает данных из модуля и порт InOut передает данные как в модуль, так и из него в зависимости от работа модуля.

Сигнал присоединяет порт одного модуля к порту другого модуля. Сигнал передает данные от одного порта к другому, как если бы порты были непосредственно связаны. Когда порт считывающий, значение сигнала, подключенного к порту, является возвращаемым. Когда порт записывающий, новое значение будет записано в сигнал, когда процесс выполнения операции записи завершится или будет приостановлен. Это сделано, чтобы все операции в рамках процесса работали с тем же значением сигнала. Это должно предотвратить то, что некоторые процессы наблюдают старое значение, в то время как другие процессы наблюдают новое значение сигнала во время выполнения. Все процессы, выполняемые в течение временного шага, будут видеть старое значение сигнала. Эта семантика сигналов такая же, как операции с сигналами в VHDL и отсроченное поведение присваивания в Verilog.

Порты всегда связаны с сигналом за исключением одного частного случая, когда порт привязывается непосредственно к другому порту. Порты всегда связаны только с одним сигналом. Этот сигнал может быть комплексным сигналом, таким как структура, но он по-прежнему рассматривается как один сигнал. Соединение сигналами происходит во время конкретизации модуля.

При построении иерархической структуры проектирования модули инстанцируются в пределах других модулей, чтобы сформировать иерархию дизайна. Особый случай соединения, упомянутый раньше, происходит, когда порт модуля верхнего уровня непосредственно связан с портом модуля более низкого уровня в момент создания конструкции. Это показано на рисунке ниже:

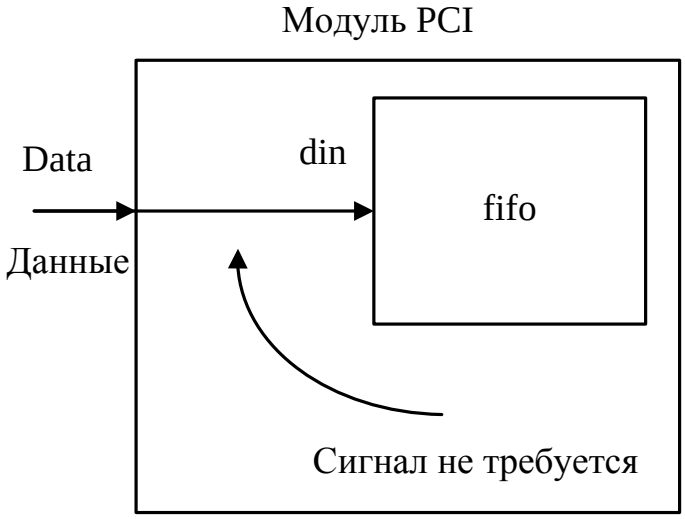


Рис. 3.11

В этом примере порт данных модуля PCI напрямую подключен к порту din модуля FIFO. Для этого случая не требуется локального сигнала.

Порты и сигналы также бывают разных размеров. Скалярные порты имеют одно измерение. Скалярный порт может быть одним из следующих типов:

C++ встроенные типы

- long
- int
- char
- short
- float
- double

SystemC типы

- sc_int<n>
- sc_uint<n>
- sc_bigint<n>
- sc_biguint<n>
- sc_bit
- sc_logic
- sc_bv<n>
- sc_lv<n>
- sc_fixed
- sc_ufixed
- sc_fix
- sc_ufix

- User defined structs

Входные, выходные и InOut порты описаны с использванием следующиго синтаксиса:

```
sc_in<porttype> // входной порт типа porttype
sc_out<porttype> // выходной порт типа porttype
sc_inout<porttype> /* ввод-вывод порт типа porttype*/
```

3.11.1. Чтение и запись портов и сигналов

Вы можете использовать методы read() и write() или оператор присваивания для чтения и записи портов и сигналов. Использование оператора присваивания делает ваш код более кратким и больше подобным коду HDL, потому что вы считываете и записываете непосредственно в портах.

Используйте read() и write() методы в явном виде для уточнения цели вашего кода, даже, если ваш код будет немного более многословным. Методы read() и write() вызываются неявным преобразованием, определенным в классе порта.

Если вам нужно неявное преобразование типов, так как тип, который вы читаете или записываете отличается от типа порта (например, если порт является BOOL и вы читаете или записываете INT), важно, чтобы вы использовали методы read() и write(). C++ автоматически применяет только одно неявное преобразование типов в любом конкретном месте, и вам нужны два неявных преобразования для чтения и записи другого типа, чем тип порта.

Не рекомендуется напрямую обращаться к портам, как для чтения, так и для записи. Типы данных порта должны использовать следующие методы для доступа к ним:

`Port_name.write ('value')`: для записи значения в порт.

`Port_name.read ()`: для чтения значения из порта

Методы write () и read () выполняют автоматическое преобразование типов из других типов данных в типы данных порта. Не всегда возможно использовать одни и те же типы данных портов или сигналов внутри процессов. При использовании разных типов данных всегда полезно использовать `port_name.read ()` и `port_name.write` («значение») для доступа к портам, и одно и то же правило применяется для сигналов.

Листинг 3.9

Программа sc_ ports_access

```
    #include <systemc.h>

    SC_MODULE(ports_access) {
      sc_in<sc_bit> a;
      sc_in<sc_bit> b;
      sc_in<bool>   en;
      sc_out<sc_lv<2> >  out;

      // Способ управления выходом
      void body () {
        /* Порты должны использовать метод read ()
для чтения значений*/
        if (en.read() == 1) {
          /* Должен использовать метод write () для
значений записи*/
          out.write(a.read() + b.read());
        }
      }
      // Способ мониторинга портов
      void monitor () {
        cout << "@" << sc_time_stamp() <<" a : " << a
          << " b : " << b << " en : " << " out : "
          << out.read() <<endl;
      }

      SC_CTOR(ports_access) {
        SC_METHOD(body);
          sensitive << a << b << en;
        SC_METHOD(monitor);
          sensitive << a << b << en << out;
      }

    };

    /* Испытательный стенд для генерации тестовых
векторов*/
    int sc_main (int argc, char* argv[]) {
      sc_signal <sc_bit> a;
      sc_signal <sc_bit> b;
      sc_signal <bool>   en;
      sc_signal <sc_lv<2> >  out;
```

```
      ports_access prt_ac("PORT_ACCESS");
        prt_ac.a(a);
        prt_ac.b(b);
        prt_ac.en(en);
        prt_ac.out(out);

      sc_start(0, SC_MS);
      // Открыть файл VCD
      sc_trace_file *wf =
sc_create_vcd_trace_file("ports_access");
        sc_trace(wf, a, "a");
        sc_trace(wf, b, "b");
        sc_trace(wf, en, "en");
        sc_trace(wf, out, "out");
      // Начало тестирования здесь
      sc_start(1, SC_MS);
      a = sc_bit('0');
      b = sc_bit('0');
      en = 1;
      sc_start(1, SC_MS);
      a = sc_bit('1');
      sc_start(1, SC_MS);
      b = sc_bit('1');
      sc_start(2, SC_MS);

      sc_close_vcd_trace_file(wf);
      return 0;// Завершить симуляцию
    }
```

Результаты моделирования:

```
@0 s a : 0 b : 0 en :  out : XX

Info: (I702) default timescale unit
@1 ms a : 0 b : 0 en :  out : XX
@1 ms a : 0 b : 0 en :  out : 00
@2 ms a : 1 b : 0 en :  out : 00
@2 ms a : 1 b : 0 en :  out : 01
@3 ms a : 1 b : 1 en :  out : 01
@3 ms a : 1 b : 1 en :  out : 10
```

На рис. 3.12 показана трассировка сигналов в GTKWave.

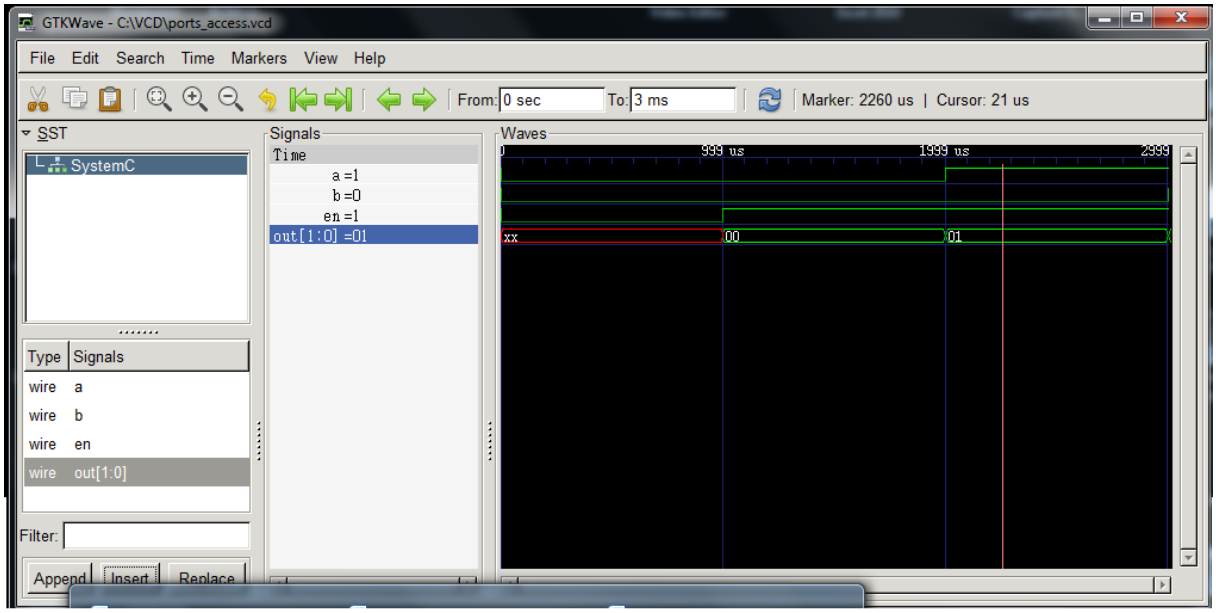


Рис. 3.12. Трассировка сигналов в GTKWave

3.11.2. Массив порты и сигналы

Для некоторых приложений может быть желательным набор портов. Например, сгенерированное компьютером описание дизайна может использовать массив портов для отображения настраиваемых размеров объектов. Чтобы объявить массив портов или сигналов, используется один и тот же синтаксис, что в C++. Пример приведен ниже:

```
sc_in<sc_logic> a[32]; /* создает порты a [0] в
[31] типа sc_logic */
```

Эта декларация создает массив портов с названиями от a[0] до a[31] типа `sc_logic`. Каждый порт должен быть индивидуально связан с портом назначения и считан.

Сигнальные массивы могут быть созданы с помощью похожего синтаксиса. Пример массива сигналов показан ниже:

```
sc_signal<sc_logic> i[16]; /* создает сигналы i
[0] - i [15] типа sc_logic */
```

Этот оператор создает массив сигналов по имени i[0] до i[15] типа `sc_logic`. Каждый сигнал должен быть индивидуально связан с портом, назначен и прочитан.

Ниже приведены примеры использования массива портов и сигналов.

Листинг 3.11

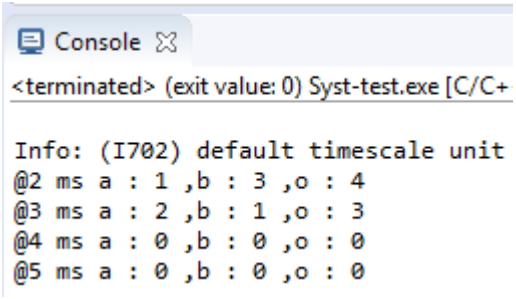
Программа sc_ports_arrays

```
#include <systemc.h>
```

```
    using namespace std;


    SC_MODULE(ports_arrays) {
      sc_in<sc_uint<2> > a[4];
      sc_in<sc_uint<2> > b[4];
      sc_out<sc_uint<3> > o[4];

      void body () {
        int i;
        for (i=0; i < 4; i ++) {
          o[i].write(a[i].read() + b[i].read());
        }
      }

      SC_CTOR(ports_arrays) {
        int j;
        SC_METHOD(body);
          for (j=0; j<4; j++) {
            sensitive << a[j] << b[j];
          }
      }
    };

    /* Испытательный стенд для генерации тестовых
векторов*/
    int sc_main (int argc, char* argv[]) {
      sc_signal<sc_uint<2> > a[4];
      sc_signal<sc_uint<2> > b[4];
      sc_signal<sc_uint<3> > o[4];

      int z;

      ports_arrays prt_ar("PORT_ARRAY");
        for (z=0; z<4; z++) {
          prt_ar.a[z](a[z]);
          prt_ar.b[z](b[z]);
          prt_ar.o[z](o[z]);
        }
      sc_start(0, SC_MS);
      // Открыть VCD файл
```

```
      sc_trace_file *wf =
sc_create_vcd_trace_file("ports_arrays");
        for (z=0; z<4; z++) {
          char str[3];
          sprintf(str, "(%0d)",z);
          sc_trace(wf,a[z],"a" + string(str));
          sc_trace(wf,b[z],"b" + string(str));
          sc_trace(wf,o[z],"o" + string(str));
        }
      // Начало тестирования здесь
      sc_start(1, SC_MS);
      for (z=0; z<4; z++) {
        a[z] = rand();
        b[z] = rand();
        sc_start(1);
        cout << "@" << sc_time_stamp() <<" a : " <<
a[z]
          << " ,b : " << b[z] << " ,o : " << o[z] <<
endl;
      }
      sc_start(2, SC_MS);

      sc_close_vcd_trace_file(wf);
      return 0;// Завершить симуляцию
    }
```

Результаты моделирования



3.11.3. Привязка сигналов

Каждый порт привязывается к одному сигналу. При чтении порта переменная, назначенная к порту, должна иметь тот же тип, что и тип порта. Для примера порт типа `sc_logic` не может быть прочитан в `int` переменной или сигналом.

Когда порты связаны с другими сигналами или портами, оба типа должны совпадать.

Ниже показан порт, связанный с другим портом (частный случай), и порт связанный с сигналом.

```
// statemach.h
#include "systemc.h"
SC_MODULE(state_machine) {
sc_in<sc_logic> clock;
sc_in<sc_logic> en;
sc_out<sc_logic> dir;
sc_out<sc_logic> status;
//  другие объявления модуля
};

// controller.h
#include "statemach.h"
SC_MODULE(controller) {
sc_in<sc_logic> clk;
sc_out<sc_logic> count;
sc_in<sc_logic> status;
sc_out<sc_logic> load;
sc_out<sc_logic> clear
sc_signal<sc_logic> lstat;
sc_signal<sc_logic> down;
state_machine *s1;
SC_CTOR(controller) {
//.. другие объявления модуля
s1 = new state_machine ("s1");
s1->clock(clk); /* специальный порт для привязки
портов*/
s1->en(lstat); /* порт, привязанный к сигналу
lstat*/
s1->dir(down); // порт, связанный с сигналом down
s1->st(status); /* специальный порт для привязки
портов*/ }
};
```

В этом примере показан модуль контроллера с некоторым числом входных и выходных портов. Модуль также включает в себя локальные сигналы `lstat` и `down`. Модуль контроллера инстанцирует модуль `state_machine` с меткой `s1`.

Модуль `state_machine` должен содержать следующие операторы.

Первый оператор:

```
s1->clock(clk);
```

связывает тактирование порта с меткой s1 с внешним портом CLK контроллера. Это пример специального случая связывания, в котором порт привязывается непосредственно к другому порту вместо сигнала. Вторая привязка порта показана ниже:

```
s1->en(lstat);
```

Этот оператор привязывает порт с меткой `s1` к локальному сигналу `lstat.` Это является примером «именнованных соединений»-Named Mapping из раздела "Позиционные соединения".

Программа `sc_signal_bind` на листинге 3.12 иллюстрирует вопросы привязки портов.

Листинг 3.12

Программа sc_signal_bind

```
#include <systemc.h>

SC_MODULE (some_block) {
  sc_in<bool>   clock;
  sc_in<sc_bit> data;
  sc_in<sc_bit> reset;
  sc_in<sc_bit> inv;
  sc_out<sc_bit> out;

  void body () {
    if (reset.read() == 1) {
      out = sc_bit(0);
    } else if (inv.read() == 1) {
      out = ~out.read();
    } else {
      out.write(data.read());
    }
  }

  SC_CTOR(some_block) {
    SC_METHOD(body);
      sensitive << clock.pos();
  }
};


SC_MODULE (signal_bind) {
```

```
      sc_in<bool> clock;

      sc_signal<sc_bit> data;
      sc_signal<sc_bit> reset;
      sc_signal<sc_bit> inv;
      sc_signal<sc_bit> out;
      some_block *block;
      int  done;

      void do_test() {
        while (true) {
          wait();
          if (done == 0) {
            cout << "@" << sc_time_stamp() <<"
Starting test"<<endl;
            wait();
            wait();
            inv = sc_bit('0');
            data = sc_bit('0');
            cout << "@" << sc_time_stamp() <<"
Driving 0 all inputs"<<endl;
            // Assert reset
            reset = sc_bit('1');
            cout << "@" << sc_time_stamp() <<"
Asserting reset"<<endl;
            // Deassert reset
            wait();
            wait();
            reset = sc_bit('0');
            cout << "@" << sc_time_stamp() <<" De-
Asserting reset"<<endl;
            // Assert data
            wait();
            wait();
            cout << "@" << sc_time_stamp() <<"
Asserting data"<<endl;
            data = sc_bit('1');
            wait();
            wait();
            cout << "@" << sc_time_stamp() <<"
Asserting inv"<<endl;
            inv = sc_bit('1');
```

```
            wait();
            wait();
            cout << "@" << sc_time_stamp() <<" De-
Asserting inv"<<endl;
            inv = sc_bit('0');
            wait();
            wait();
            done = 1;
          }
        }
      }

      void monitor() {
        cout << "@" << sc_time_stamp() << " data :"
<< data
          << " reset :" << reset << " inv :"
          << inv << " out :" << out <<endl;
      }

      SC_CTOR(signal_bind) {
        block = new some_block("SOME_BLOCK");
        block->clock       (clock)  ;
        block->data        (data)   ;
        block->reset       (reset)  ;
        block->inv         (inv)    ;
        block->out         (out)    ;
        done = 0;

        SC_CTHREAD(do_test,clock.pos());
        SC_METHOD(monitor);
          sensitive << data << reset << inv << out;
      }
    };

    int sc_main (int argc, char* argv[]) {
      sc_signal<bool> clock;
      int i;

      signal_bind  object("SIGNAL_BIND");
        object.clock (clock);
      sc_trace_file *wf =
sc_create_vcd_trace_file("signal_bind");
```

```
    sc_trace(wf, object.clock, "clock");
    sc_trace(wf, object.reset, "reset");
    sc_trace(wf, object.data, "data");
    sc_trace(wf, object.inv, "inv");
    sc_trace(wf, object.out, "out");

  sc_start(0, SC_MS);
  for(i=0;i<100;i++) {
    if (object.done == 0) {
      clock = 0;
      sc_start(1, SC_MS);
      clock = 1;
      sc_start(1, SC_MS);
    }
  }
  sc_close_vcd_trace_file(wf);

  cout<<"Terminating Simulation"<<endl;
  return 0;// Завершить симуляцию
}
```

Результаты моделирования:

```
<terminated> (exit value: 0) Syst-test.exe [C/C++ Application] C:\SystemC\workspace\Syst-test\Debu

Info: (I702) default timescale unit used for tracing: 1 ps (signal_bind.vcd)
@3 ms Starting test
@7 ms Driving 0 all inputs
@7 ms Asserting reset
@7 ms data :0 reset :1 inv :0 out :0
@11 ms De-Asserting reset
@11 ms data :0 reset :0 inv :0 out :0
@15 ms Asserting data
@15 ms data :1 reset :0 inv :0 out :0
@17 ms data :1 reset :0 inv :0 out :1
@19 ms Asserting inv
@19 ms data :1 reset :0 inv :1 out :1
@21 ms data :1 reset :0 inv :1 out :0
@23 ms De-Asserting inv
@23 ms data :1 reset :0 inv :0 out :1
```

Трассировка сигналов показана на рис. 3.13

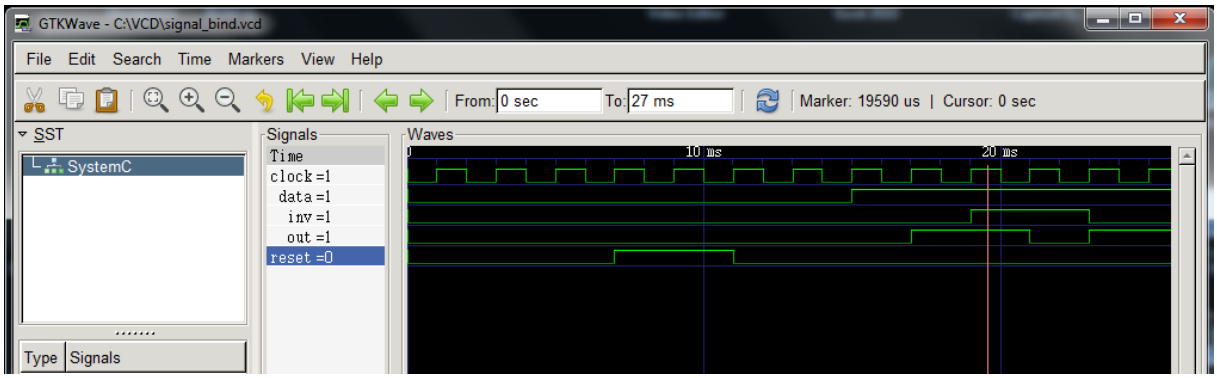


Рис. 3.13. Трассировка сигналов программы SC_SIGNAL_BIND

3.11.4. Разрешенные логические векторы

В реальном оборудовании бывают ситуации, когда нам нужно моделировать шину с тремя состояниями. Таким образом, несколько устройств могут управлять одной и той же шиной. Есть шина адреса и шина данных.

Чтобы моделировать это, SystemC предоставляет разрешенные логические векторы. В устройстве, котором управляется 0 и 1, выигрывает шина с 1. На приведенном ниже рисунке 3.14 у нас есть модуль A, выдающий 1, и модуль B, управляющий Z, т.е. отключенное состояние. Сигнал разрешения Y получает 1.

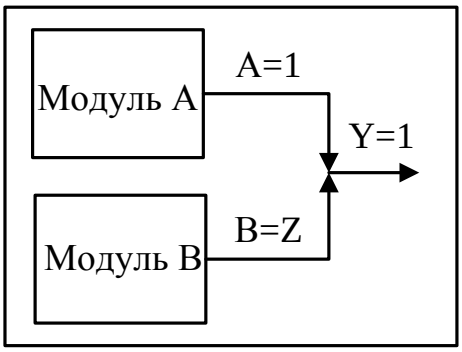


Рис. 3.14

Когда модуль A выдает 1, а модуль B выдает 0, тогда на выходе Y появляется X. X обозначает неизвестное значение.

Листинг 3.13

```
#include <systemc.h>

SC_MODULE(module_A) {
  sc_in_rv<1>  in;
  sc_out_rv<1> out;
  sc_inout_rv<4> inout;

  void body () {
```

```
    out.write(in);
    if (in.read() == 1) {
      out.write(1);

      inout.write(rand());
    } else {
      out.write('z');
      inout.write('zzzz');
    }
  }

  SC_CTOR(module_A) {
    SC_METHOD(body);
      sensitive << in;
  }
};
```

3.11.5. Разрешенные векторные сигналы

Разрешенные векторные сигналы обладают тем же свойством, что и разрешенные логические векторы. Разрешенные векторные сигналы используются для соединения двух разрешенных портов логического вектора. Одна из ключевых проблем заключается в том, что разрешенные векторные сигналы не должны выходить за пределы процесса / метода.

Листинг 3.14

Программа sc_vector_signal

```
//www.asic-world.com
#include <systemc.h>

SC_MODULE(resolve) {
  sc_in_rv<1>  in;
  sc_out_rv<1> out;
  sc_inout_rv<4> inout;

  void body () {
    out.write(in);
    if (in.read() == 1) {
      out.write(1);
      inout.write(rand());
```

```
        } else {
          out.write("z");
          inout.write("zzzz");
        }
      }

      SC_CTOR(resolve) {
        SC_METHOD(body);
          sensitive << in;
      }
    };

    /* Испытательный стенд для генерации тестовых
векторов*/
    int sc_main (int argc, char* argv[]) {
      sc_signal_rv<1> in1,in2;
      sc_signal_rv<1> out;
      sc_signal_rv<4> inout;

      resolve rs1("RESOLVE1");
        rs1.in(in1);
        rs1.out(out);
        rs1.inout(inout);
      resolve rs2("RESOLVE2");
        rs2.in(in2);
        rs2.out(out);
        rs2.inout(inout);

      sc_start(0, SC_MS);
      // Открыть VCD file
      sc_trace_file *wf =
sc_create_vcd_trace_file("resolve");
        sc_trace(wf, in1, "in1");
        sc_trace(wf, in2, "in2");
        sc_trace(wf, out, "out");
        sc_trace(wf, inout, "inout");
      // Начало тестирования здесь
      cout << "@" << sc_time_stamp() <<" in1 : " <<
in1
        <<" in2 : " << in2 <<" out : " << out <<"
inout : " << inout << endl;
      sc_start(1, SC_MS);
```

```
      in1 = 0;
      in2 = 0;
      cout << "@" << sc_time_stamp() <<" in1 : " <<
in1
        <<" in2 : " << in2 <<" out : " << out <<"
inout : " << inout << endl;
      sc_start(1, SC_MS);
      in1 = 1;
      cout << "@" << sc_time_stamp() <<" in1 : " <<
in1
        <<" in2 : " << in2 <<" out : " << out <<"
inout : " << inout << endl;
      sc_start(1, SC_MS);
      in1 = 0;
      in2 = 1;
      cout << "@" << sc_time_stamp() <<" in1 : " <<
in1
        <<" in2 : " << in2 <<" out : " << out <<"
inout : " << inout << endl;
      sc_start(1, SC_MS);
      in2 = 0;
      cout << "@" << sc_time_stamp() <<" in1 : " <<
in1
        <<" in2 : " << in2 <<" out : " << out <<"
inout : " << inout << endl;
      sc_start(1, SC_MS);
      in1 = 1;
      in2 = 1;
      cout << "@" << sc_time_stamp() <<" in1 : " <<
in1
        <<" in2 : " << in2 <<" out : " << out <<"
inout : " << inout << endl;
      sc_start(1, SC_MS);
      in1 = 0;
      in2 = 0;
      cout << "@" << sc_time_stamp() <<" in1 : " <<
in1
        <<" in2 : " << in2 <<" out : " << out <<"
inout : " << inout << endl;
      sc_start(2, SC_MS);
      cout << "@" << sc_time_stamp() <<" in1 : " <<
in1
```

```
        <<" in2 : " << in2 <<" out : " << out <<"
inout : " << inout << endl;
      sc_close_vcd_trace_file(wf);
      return 0;// Завершить симуляцию
    };
```

Результыты моделирования:

```
@0 s in1 : X in2 : X out : Z inout : ZZZZ

Info: (I702) default timescale unit used for
@1 ms in1 : X in2 : X out : Z inout : ZZZZ
@2 ms in1 : 0 in2 : 0 out : Z inout : ZZZZ
@3 ms in1 : 1 in2 : 0 out : 1 inout : 1101
@4 ms in1 : 0 in2 : 1 out : 1 inout : 1111
@5 ms in1 : 0 in2 : 0 out : Z inout : ZZZZ
@6 ms in1 : 1 in2 : 1 out : 1 inout : XXXX
@8 ms in1 : 0 in2 : 0 out : Z inout : ZZZZ
```

Трассировка сигналов показана на рис. 3.15.

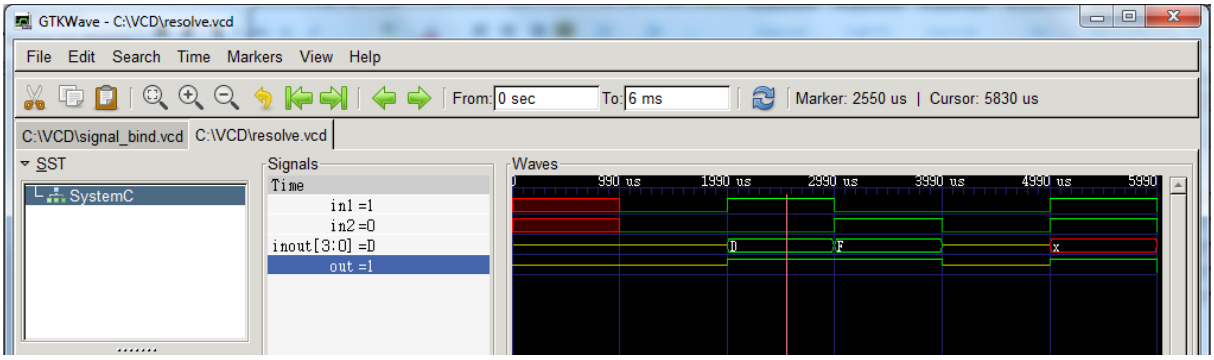


Рис. 3.15. Трассировка сигналов

3.12. Тактирование

Объекты Clock являются специальными объектами в SystemC. Они генерируют сигналы синхронизации, используемые, чтобы синхронизировать события в симуляции. Вызов событий clock во времени требуется, чтобы параллельные события в аппаратных средствах правильно моделировались симулятором на последовательном компьютере. Объект clock имеет ряд элементов данных для хранения настроек, а также методов для выполнения тактирования. Для создания объекта синхронизации используйте следующий синтаксис:

```
sc_clock CLOCK1 ("CLOCK1", 20, 0,5, 2, true);
```

Эта декларация создаст объект с именем clock с периодом 20 единиц, рабочий цикл 50%, первый фронт будет происходить на 2-ой единице времени, и первое значение будет `true` (1). Все эти аргументы имеют значения по умолчанию, за исключением имени `clock`. По умолчанию

период равен 1, рабочий цикл до 0,5, первый фронт на «0», а первое значение `true`.

Обычно `Clock` создается на верхнем уровне дизайна в Testbench и опускается вниз по иерархии модулей для остальной части конструкции. Это позволяет синхронизировать все участки конструкции или всю конструкцию одними тактами.

В примере из проекта с модулем `SC_MODULE (filter)` программа `sc_main` конструкции создает `clock` и соединяет его со всеми компонентами основного модуля:

```
int sc_main(int argc, char*argv[]) {
sc_signal<int> val;
sc_signal<sc_logic> load;
sc_signal<sc_logic> reset;
sc_signal<int> result;
sc_clock ck1("ck1", 20, 0.5, 0, true);
filter f1("filter");
f1.clk(ck1.signal());
f1.val(val);
f1.load(load);
f1.reset(reset);
f1.out(result);
// остальная часть sc_main не показана
}
```

В этом примере процедура `sc_main` верхнего уровня конкретизирует модуль с названием фильтр и объявляет некоторые локальные сигналы, которые будут подключать фильтр к другим конкретным модулям. Обратите внимание на то, что сигнал CLK не объявлен, вместо этого, чтобы объект CLK инстанциировался, его параметры настройки установливаются, и его `signal_method` используется для обеспечения сигнала синхронизации. Функция `ck1.signal ()` отображается на CLK порт объекта фильтра. В этом примере `clock` называют CK1 и тактовая частота определяется как 20 единиц измерения времени. Каждые 20 единиц времени `clock` сделают полный переход от `true` к `false` и обратно.

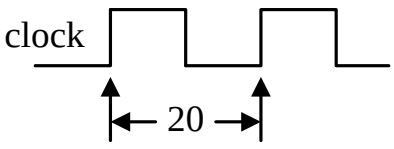


Рис. 3.16

Для модуля тактирования заданного так:

```
sc_clock ck1("ck1", 20, 0.5, 2, true);
```

получим:

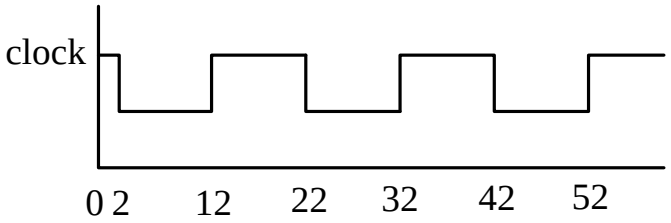


Рис. 3.17

При привязке такта к порту дизайнер должен использовать тактовый сигнал, который генерируется объектом `clock` в обозначении порта. Это делается с помощью метода сигнала, характеризующего объект `clock`. Обратите внимание на то, что CLK порта фильтра отображается на ck1.signal (). Этот тактовый сигнал генерируется объектом синхронизации.

Для процессов SC_CTHREAD `clock` объект отображен непосредственно на тактовом входе процесса и `signal_method` не требуется. Часто требуется указать размерность времени (например, SC_MS).

При привязке тактов к порту разработчик должен использовать тактовый сигнал, генерируемый тактируемым объектом. Это делается с использованием метода сигнала объекта clock.

Ниже приведен пример, в котором мы заменили тактирование вручную сигналом `sc_clock` и использовали метод `clock.signal ()` при привязке порта.

Листинг 3.15

Программа sc_clock

```
#include <systemc.h>

SC_MODULE (some_block) {
  sc_in<bool>   clock;
  sc_in<sc_bit> data;
  sc_in<sc_bit> reset;
  sc_in<sc_bit> inv;
  sc_out<sc_bit> out;

  void body () {
    if (reset.read() == 1) {
      out = sc_bit(0);
    } else if (inv.read() == 1) {
      out = ~out.read();
    } else {
```

```
        out.write(data.read());
      }
    }

    SC_CTOR(some_block) {
      SC_METHOD(body);
        sensitive << clock.pos();
    }
  };


  SC_MODULE (signal_bind) {
    sc_in<bool> clock;

    sc_signal<sc_bit> data;
    sc_signal<sc_bit> reset;
    sc_signal<sc_bit> inv;
    sc_signal<sc_bit> out;
    some_block *block;
    int  done;

    void do_test() {
      while (true) {
        wait();
        if (done == 0) {
          cout << "@" << sc_time_stamp() <<"
Starting test"<<endl;
          wait();
          wait();
          inv = sc_bit('0');
          data = sc_bit('0');
          cout << "@" << sc_time_stamp() <<"
Driving 0 all inputs"<<endl;
          // Установить сброс
          reset = sc_bit('1');
          cout << "@" << sc_time_stamp() <<"
Asserting reset"<<endl;
          // Отменить сброс
          wait();
          wait();
          reset = sc_bit('0');
```

```
            cout << "@" << sc_time_stamp() <<" De-
Asserting reset"<<endl;
            // Установить data
            wait();
            wait();
            cout << "@" << sc_time_stamp() <<"
Asserting data"<<endl;
            data = sc_bit('1');
            wait();
            wait();
            cout << "@" << sc_time_stamp() <<"
Asserting inv"<<endl;
            inv = sc_bit('1');
            wait();
            wait();
            cout << "@" << sc_time_stamp() <<" De-
Asserting inv"<<endl;
            inv = sc_bit('0');
            wait();
            wait();
            done = 1;
          }
        }
      }

      void monitor() {
        cout << "@" << sc_time_stamp() << " data :"
<< data
          << " reset :" << reset << " inv :"
          << inv << " out :" << out <<endl;
      }

      SC_CTOR(signal_bind) {
        block = new some_block("SOME_BLOCK");
        block->clock       (clock)  ;
        block->data        (data)   ;
        block->reset       (reset)  ;
        block->inv         (inv)    ;
        block->out         (out)    ;
        done = 0;

        SC_CTHREAD(do_test,clock.pos());
```

```
        SC_METHOD(monitor);
          sensitive << data << reset << inv << out;
      }

    };

    int sc_main (int argc, char* argv[]) {
      sc_clock clock ("my_clock",1,0.5);

      signal_bind  object("SIGNAL_BIND");
        object.clock (clock);
      sc_trace_file *wf =
sc_create_vcd_trace_file("clock");
        sc_trace(wf, object.clock, "clock");
        sc_trace(wf, object.reset, "reset");
        sc_trace(wf, object.data, "data");
        sc_trace(wf, object.inv, "inv");
        sc_trace(wf, object.out, "out");

      sc_start(100);
      sc_close_vcd_trace_file(wf);

      cout<<"Terminating Simulation"<<endl;
      return 0;// Завершить симуляцию
    }
```

Результаты моделирования

```
Info: (I702) default timescale unit used
@1 ns Starting test
@3 ns Driving 0 all inputs
@3 ns Asserting reset
@3 ns data :0 reset :1 inv :0 out :0
@5 ns De-Asserting reset
@5 ns data :0 reset :0 inv :0 out :0
@7 ns Asserting data
@7 ns data :1 reset :0 inv :0 out :0
@8 ns data :1 reset :0 inv :0 out :1
@9 ns Asserting inv
@9 ns data :1 reset :0 inv :1 out :1
@10 ns data :1 reset :0 inv :1 out :0
@11 ns De-Asserting inv
@11 ns data :1 reset :0 inv :0 out :1
```

3.13. Время

Различие между HDL и языком программирования SystemC - это понятие времени и совпадения.

Рассмотрим тип данных времени. Этот тип данных включает:

```
Sc_start ()
Sc_time_stamp ()
Wait (sc_time)
Sc_simulation_time ()
sc_event
```

`Sc_time` - специальный тип данных, который используется для представления временных задержек и временных интервалов моделирования, включая задержки и тайм-ауты. Объект класса `sc_time` строится из `double` и `sc_time_unit`. Время должно быть представлено внутренне как целое число без знака, по меньшей мере, 64 бита. Как и любой другой тип данных в SystemC, sc_time также позволяет выполнять арифметические операции.

Типы перечисления, которые обозначают различные единицы времени, приведены ниже

```
SC_FS = 0
SC_PS = 1
SC_NS = 2
SC_US = 3
SC_MS = 4
SC_SEC = 5
```

SC_SET_TIME_RESOLUTION

Время должно быть представлено внутренне как целое, кратное временному разрешению. Временное разрешение по умолчанию составляет 1 пикосекунду. Временное разрешение можно изменить только вызовом функции sc_set_time_resolution. Эта функция вызывается только во время разработки, вызывается не более одного раза и не вызывается после создания объекта типа sc_time с ненулевым значением времени. Функция sc_get_time_resolution должна вернуть разрешение по времени.

SC_ZERO_TIME

SC_ZERO_TIME представляет время задержки 0. Хорошей практикой является использование этой константы при записи нулевого значения времени, например, при создании дельта-уведомления или дельта-тайм-аута.

Листинг 3.16

Программа SC_TIME_RESOL

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
```

```
      /* sc_set_time_resol ... должен быть вызван до
объявления переменной sc_time */
      sc_set_time_resolution(1,SC_PS);
      // Объявить переменные sc_time
      sc_time             t1(10,SC_NS);
      sc_time             t2(5,SC_PS);
      sc_time             t3,t4(1,SC_PS),t5(1,SC_PS);
      // Печать всех переменных
      cout <<"Value of t1 "<< t1.to_string() << endl;
      cout <<"Value of t2 "<< t2.to_string() << endl;
      cout <<"Value of t3 "<< t3.to_string() << endl;
      cout <<"Value of t4 "<< t4.to_string() << endl;
      cout <<"Value of t5 "<< t5.to_string() << endl;
      // Запуск симуляции
      sc_start(0, SC_MS);
      sc_start(1, SC_MS);
      // Получить текущее время
      t3 = sc_time_stamp();
      cout <<"Value of t3 "<< t3.to_string() << endl;
      // Запустите еще некоторое время
      sc_start(20, SC_MS);
      // Получить текущее время
      t4 = sc_time_stamp();
      // Печать переменных для нового времени
      cout <<"Value of t4 "<< t4.to_string() << endl;
      // Так вы делаете операцию вычитания времени
      t5 = t4 - t3;
      cout <<"Value of t5 "<< t5.to_string() << endl;
      // Так мы сравниваем операцию
      if (t5 > t2) {
        cout <<" t5 is greated then t2" << endl;
      } else {
        cout <<" t2 is greated then t5" << endl;
      }
      return 1;
    }
```

Результаты моделирования:

```
Value of t1 10 ns
Value of t2 5 ps
Value of t3 0 s
Value of t4 1 ps
Value of t5 1 ps
Value of t3 1 ms
Value of t4 21 ms
Value of t5 20 ms
 t5 is greated then t2
```

SC_START

Sc_start () является ключевым методом в SystemC. Этот метод запускает фазу моделирования, которая состоит из инициализации и выполнения. Методы sc_start () в разных ситуациях выполняют операции, перечисленные ниже.

- Вызывается в первый раз: sc_start запускает планировщик, который запускается до времени моделирования, переданного в качестве аргумента (если передан аргумент).
- Когда вызывается во второй и последующие моменты, sc_start возобновит работу планировщика с момента, который он достиг в конце предыдущего вызова sc_start. Планировщик будет выполняться за время, пришедшее в качестве аргумента (если был передан аргумент), относительно текущего времени моделирования.
- Когда в качестве аргумента передается время, планировщик будет выполнять моделирование «до» и, включив фазу уведомляемого времени, которая увеличивает время моделирования, будет действовать до конечного момента времени (рассчитывается путем добавления времени, указанного в качестве аргумента в момент моделирования, когда вызывается функция sc_start).
- Вызывается без аргументов: планировщик будет запускаться до тех пор, пока он не завершится.
- Вызывается с нулевым аргументом времени: планировщик должен запускаться для одного дельта-цикла.

После запуска планировщик будет запущен до тех пор, пока моделирование не завершится, или приложение вызовет функцию sc_stop, или произойдет другое исключение. Как только функция sc_stop была вызвана, функция sc_start больше не вызывается. Функция sc_start может вызываться из функции sc_main, и только из функции sc_main.

Листинг 3.17

Программа sc_start

```
#include <systemc.h>
```

```
    SC_MODULE (some_block) {
      sc_in<bool>   clock;
      sc_in<sc_bit> data;
      sc_in<sc_bit> reset;
      sc_in<sc_bit> inv;
      sc_out<sc_bit> out;

      void body () {
        if (reset.read() == 1) {
          out = sc_bit(0);
        } else if (inv.read() == 1) {
          out = ~out.read();
        } else {
          out.write(data.read());
        }
      }

      SC_CTOR(some_block) {
        SC_METHOD(body);
          sensitive << clock.pos();
      }
    };


    SC_MODULE (signal_bind) {
      sc_in<bool> clock;

      sc_signal<sc_bit> data;
      sc_signal<sc_bit> reset;
      sc_signal<sc_bit> inv;
      sc_signal<sc_bit> out;
      some_block *block;

      void do_test() {
        while (true) {
          wait();
          cout << "@" << sc_time_stamp() <<" Starting
test"<<endl;
          wait();
          wait();
          inv = sc_bit('0');
          data = sc_bit('0');
```

```
          cout << "@" << sc_time_stamp() <<" Driving
0 all inputs"<<endl;
          // Установить сброс
          reset = sc_bit('1');
          cout << "@" << sc_time_stamp() <<"
Asserting reset"<<endl;
          // Отменить сброс
          wait();
          wait();
          reset = sc_bit('0');
          cout << "@" << sc_time_stamp() <<" De-
Asserting reset"<<endl;
          // Установить данные
          wait();
          wait();
          cout << "@" << sc_time_stamp() <<"
Asserting data"<<endl;
          data = sc_bit('1');
          wait();
          wait();
          cout << "@" << sc_time_stamp() <<"
Asserting inv"<<endl;
          inv = sc_bit('1');
          wait();
          wait();
          cout << "@" << sc_time_stamp() <<" De-
Asserting inv"<<endl;
          inv = sc_bit('0');
          wait();
          wait();
          cout<<"Terminating Simulation"<<endl;
          sc_stop(); /* sc_stop запускает завершение
моделирования*/
        }
      }

      void monitor() {
        cout << "@" << sc_time_stamp() << " data :"
<< data
          << " reset :" << reset << " inv :"
          << inv << " out :" << out <<endl;
      }
```

```
      SC_CTOR(signal_bind) {
        block = new some_block("SOME_BLOCK");
        block->clock       (clock)  ;
        block->data        (data)   ;
        block->reset       (reset)  ;
        block->inv         (inv)    ;
        block->out         (out)    ;

        SC_CTHREAD(do_test,clock.pos());
        SC_METHOD(monitor);
          sensitive << data << reset << inv << out;
      }
    };

    int sc_main (int argc, char* argv[]) {
      sc_clock clock ("my_clock",1,0.5);

      signal_bind  object("SIGNAL_BIND");
        object.clock (clock);
      sc_trace_file *wf =
sc_create_vcd_trace_file("sc_start");
        sc_trace(wf, object.clock, "clock");
        sc_trace(wf, object.reset, "reset");
        sc_trace(wf, object.data, "data");
        sc_trace(wf, object.inv, "inv");
        sc_trace(wf, object.out, "out");
      sc_start(0, SC_MS); /* Первый раз вызывается
планировщик инициализации*/
      sc_start(1, SC_MS); /* Моделирование приращения
на единицу времени*/
      sc_start();  /* Запуск симуляции до появления
sc_stop*/
      /*sc_start () завершается и возвращается к
следующему запуску после встречи sc_stop ()*/
      cout<<"Closing VCD File"<<endl;
      sc_close_vcd_trace_file(wf);
      return 0;// Завершение симуляции
    }
```

Результаты моделирования

```
@1 ns Starting test
@3 ns Driving 0 all inputs
@3 ns Asserting reset
@3 ns data :0 reset :1 inv :0 out :0
@5 ns De-Asserting reset
@5 ns data :0 reset :0 inv :0 out :0
@7 ns Asserting data
@7 ns data :1 reset :0 inv :0 out :0
@8 ns data :1 reset :0 inv :0 out :1
@9 ns Asserting inv
@9 ns data :1 reset :0 inv :1 out :1
@10 ns data :1 reset :0 inv :1 out :0
@11 ns De-Asserting inv
@11 ns data :1 reset :0 inv :0 out :1
Terminating Simulation

Info: /OSCI/SystemC: Simulation stopped by user.
```

SC_TIME_STAMP()

Функция sc_time_stamp возвращает текущее время моделирования. Во время разработки и инициализации функция вернет нулевое значение.

Листинг 3.18

Программа sc_time_stamp

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  cout<<"Current time is "<< sc_time_stamp() <<
endl;
  sc_start(1);
  cout<<"Current time is "<< sc_time_stamp() <<
endl;
  sc_start(100);
  cout<<"Current time is "<< sc_time_stamp() <<
endl;
  sc_stop();
  cout<<"Current time is "<< sc_time_stamp() <<
endl;
  return 0;// Завершение симуляции
}
```

Результаты моделирования

```
<terminated> (exit value: 0) Syst-test.exe [C/C++ Application] C:\SystemC\
Current time is 0 s
Current time is 1 ms
Current time is 101 ms

Info: /OSCI/SystemC: Simulation stopped by user.
Current time is 101 ms
```

3.14. События

Многие мероприятия в реальной системе происходят одновременно или параллельно. SystemC для моделирования процессов использует параллелизм. В большинстве управляемых событиями симуляторов параллелизм не является истинным параллельным выполнением. На самом деле, параллельное моделирование работает подобно кооперативной многозадачности. Каждый процесс в симуляторе выполняет небольшой кусок кода, а затем добровольно отдает управление, чтобы другие процессы выполнялись в том же моделируемом временном пространстве.

Ядро симулятора отвечает за запуск процессов и управление тем, какой процесс выполняется следующим. Из-за кооперативного характера симулятора, процессы должны приостановливать себя, чтобы позволить выполнение других параллельных процессов.

SystemC в настоящее время предусматривает два основных типа процессов: SC_THREAD процессы и процессы SC_METHOD. Третий тип SC_CTHREAD, имеет незначительные изменения в процессе SC_THREAD и допускает тактирование.

Переключением выпполняемых процессов могут управлять *события(event)*

SystemC использует класс sc_event для моделирования событий. Этот класс позволяет выполнить явный запуск или переключение процессов от событий с помощью метода уведомления.

ОПРЕДЕЛЕНИЕ: Событие в SystemC – это появление sc_event уведомления, которое происходит в один момент времени. Событие не имеет длительность или значения.

Событие – это нечто, происходящее в определённое время. У события нет никакого значения и продолжительности, оно либо произошло, либо ещё нет. В SystemC событие определяется с помощью класса sc_event. Процессы могут реагировать на события, но для этого в списке чувствительности процесса должно быть явно указано событие, к которому он будет чувствителен, например,

```
sc_event delay; …
SC_METHOD(do_delay);
sensitive << delay;
```

Событие можно вызвать с помощью метода `notify()`, в качестве параметра может быть указан промежуток времени в формате `sc_time`, через которое событие должно произойти (в таком случае, событие будет запланированным), например,

```
// Событие произойдет немедленно delay.notify();
// Событие произойдет через промежуток времени 0
delay.notify(SC_ZERO_TIME);
```

```
// Событие произойдет через промежуток времени 20 нс (запланированное событие)
delay.notify(20, SC_NS);
```

Отменить вызов запланированного события можно с помощью метода `cancel(): delay.cancel();`

Ниже приведен пример модели логического элемента НЕ7404, задержка срабатывания которого реализовано с помощью события:

```
SC_MODULE(not)
{ sc_in <bool> A;
sc_out <bool> F;
 sc_event delay;
SC_CTOR(not)
{
SC_METHOD(do_delay);
sensitive << A;
SC_METHOD(do_not);
sensitive << delay;
}
void do_delay()
{
delay.notify(22, SC_NS);
}
void do_not(){F.write(!A.read());}
};
```

В данном примере временная задержка реализована с использованием события. Событие `delay,` объявленное в теле модуля происходит тогда, когда на входе элемента НЕ меняется входной сигнал `(sc_in <bool> A;),` метод `do_delay` чувствителен к изменению сигнала А. Команда `delay.notify(22, SC_NS)` активизирует событие `delay` спустя 22 ns, в свою очередь метод `do_not,` чувствительный к событию delay, реагирует и меняет значение на выходе элемента НЕ.

Для того, чтобы поймать событие, надо наблюдать за ним. SystemC позволяет процессам ждать событие, используя динамическую или статическую чувствительность. Если событие происходит, а никакие процессы не ждут, чтобы поймать его, это событие проходит незамеченным.

Синтаксис для объявления события:

`sc_event event_name`$_1$`[, event name`$_i$`]...;`

Тип события `sc_event` обеспечивает следующие функциональные возможности.

Конструктор - Объект «события» может быть создан путем вызова конструктора без любых аргументов.

Например:

```
sc_event my_event;
```

`Notify` – о событие может быть уведомлено путем вызова метода уведомления объекта события. Например:

```
my_event.notify(); // немедленно уведомлять
my_event.notify( SC_ZERO_TIME ); /* уведомлять следующий дельта-цикл*/
my_event.notify( 10, SC_NS );/* уведомлять через 10 нс*/
sc_time t( 10, SC_NS );
my_event.notify( t ); // то же самое
```

Кроме того, функции позволяют иметь функциональную нотацию для уведомления о событии. Например:

```
notify( my_event ); // немедленно уведомлять
notify( SC_ZERO_TIME, my_event ); /* уведомлять следующий дельта-цикл*/
notify( 10, SC_NS, my_event ); /* уведомлять через 10 нс*/
sc_time t( 10, SC_NS );
notify( t, my_event ); // то же самое
```

`Cancel` - уведомление о событии может быть отменено путем вызова `Cancel` - метода () объекта события. Например:

```
my_event.cancel(); // отменить опоздание
```

Конструктор копирования и оператор присваивания типа события отключены.

Каналами можно построить любое количество объектов событий - по одному для каждого типа события, которое оно может генерировать. Канал может уведомить о событии, вызвав один из методов уведомления объекта о события. Тем не менее, создание и уведомления о событиях не ограничивается каналами.

`Event_name.notify ()` - это функция-член уведомления, которая создает немедленное уведомление. Все экземпляры процесса, чувствительные к событию, должны быть запущены до того, как управление будет возвращено из функции уведомления.

`Event_name.cancel ()` является функцией-членом, которая удалит все ожидающие уведомления для этого события.

Листинг 3.19

Программа sc_event

```
    #include <systemc.h>

    SC_MODULE (events) {
      sc_in<bool> clock;

      sc_event  e1;
      sc_event  e2;

      void do_test1() {
        while (true) {
          /* Подождите, пока не появится
положительный фронт такта*/
          wait();
          cout << "@" << sc_time_stamp() <<" Starting
test"<<endl;
          // Ожидание положительного фронта такта
          wait();
          cout << "@" << sc_time_stamp() <<"
Triggering e1"<<endl;
          // Переключающее событие e1
          e1.notify(5,SC_NS);
          // Ожидание положительного фронта такта
          wait();
          // Ожидание события e2
          wait(e2);
          cout << "@" << sc_time_stamp() <<" Got
Trigger e2"<<endl;
          // Ожидание положительного фронта такта
          wait();
          cout<<"Terminating Simulation"<<endl;
          sc_stop(); /* sc_stop запускает завершение
моделирования*/
        }
      }

      void do_test2() {
        while (true) {
          // Ожидание события e2
          wait(e1);
          cout << "@" << sc_time_stamp() <<" Got
Trigger e1"<<endl;
```

```
        // Ожидание  3-х положительных фронтов тактов
wait(3);
          cout << "@" << sc_time_stamp() <<"
Triggering e2"<<endl;
          // Переключающее событие e2
          e2.notify();
        }
      }

      SC_CTOR(events) {
        SC_CTHREAD(do_test1,clock.pos());
        SC_CTHREAD(do_test2,clock.pos());
      }
    };

    int sc_main (int argc, char* argv[]) {
      sc_clock clock ("my_clock",1,0.5);

      events  object("events");
        object.clock (clock);

      sc_start(0); /* Первый раз вызывается
планировщик инициализации*/
      sc_start();  /* Запустить симуляцию до
появления sc_stop*/
      return 0;// Завершение симуляции
    }
```

Результаты моделирования

```
@1 ns Starting test
@2 ns Triggering e1
@7 ns Got Trigger e1
@10 ns Triggering e2
@11 ns Got Trigger e2
Terminating Simulation
```

3.14.1. Функция wait ()

`Wait()` приостанавливает поток или экземпляр процесса с тактированием, из которого она вызывается. В очередной раз wait() может быть вызвана только в том случае, когда экземпляр этого процесса будет возобновлен. Динамическая чувствительность определяется аргументами, переданными в функцию wait (пример: `clock`).

Вызов функции `wait` с пустым списком аргументов или с одним целым аргументом должен использовать статическую чувствительность

экземпляра процесса. Это единственная форма ожидания, разрешенная в рамках процесса с тактируемым потоком. Вызов функции `wait` с одним или несколькими нецелыми аргументами должен переопределять статическую чувствительность экземпляра процесса.

При вызове функции `wait` с параметром, переданным по ссылке, приложение должно гарантировать, что время жизни любых фактических аргументов, переданных по ссылке, увеличивается с момента вызова функции до момента завершения вызова функции и, кроме того, в случае параметра типа sc_time, приложение не должно изменять значение фактического аргумента в течение этого периода.

Примечание. Нельзя вызывать функцию `wait` из процесса метода.

`Wait` принимает параметры, некоторые из которых перечислены ниже:

`Wait ()`: дождитесь появления события чувствительного списка.

`Wait (int)`: дождитесь появления n событий; события - целое в чувствительном списке.

`Wait (event)`: дождитесь события, указанного в качестве параметра.

`Wait (double, sc_time_unit)`: подождите указанное время.

`Wait (double, sc_time_unit, event)`: дождитесь заданного времени или события.

Листинг 3.20

Программа sc_wait

```
#include <systemc.h>

SC_MODULE (wait_example) {
  sc_in<bool> clock;

  sc_event  e1;
  sc_event  e2;

  void do_test1() {
    while (true) {
      // Ждать положительного фронта такта
wait();
      cout << "@" << sc_time_stamp() <<" Starting
test"<<endl;
     // Ждать 5-ти положительных фронтов тактов
      wait(5);
```

```
          cout << "@" << sc_time_stamp() <<"
Triggering e1"<<endl;
          // Trigger event e1
          e1.notify(5,SC_NS);
          // Ждать 3 нс
          cout << "@" << sc_time_stamp() <<" Time
before wait 3 ns"<<endl;
          wait(3, SC_NS);
          cout << "@" << sc_time_stamp() <<" Time
after wait 3 ns"<<endl;
          // Ждать события e2
          wait(e2);
          cout << "@" << sc_time_stamp() <<" Got
Trigger e2"<<endl;
          // Ждать положительного фронта такта
          wait(30);
          cout<<"Terminating Simulation"<<endl;
          sc_stop(); /* sc_stop запускает завершение
моделирования*/
        }
      }

      void do_test2() {
        while (true) {
          // Ждать события e1
          wait(e1);
          cout << "@" << sc_time_stamp() <<" Got
Trigger e1"<<endl;
          // Ждать трех положительных фронтов тактов
          wait(20);
          cout << "@" << sc_time_stamp() <<"
Triggering e2"<<endl;
          // Переключение событием e2
          e2.notify();
          // Ждать или 20нс или события e1
          wait(20,SC_NS,e1);
          cout << "@" << sc_time_stamp() <<" Done
waiting for 20ns or event e1"<<endl;
        }
      }

      SC_CTOR(wait_example) {
```

```
      SC_CTHREAD(do_test1,clock.pos());
      SC_CTHREAD(do_test2,clock.pos());
    }
};

int sc_main (int argc, char* argv[]) {
  sc_clock clock ("my_clock",1,0.5);

  wait_example  object("wait");
    object.clock (clock);

    sc_start(0); /* Первый раз вызывается
планировщик инициализации*/
    sc_start();  /* Запустить симуляцию до
появления sc_stop*/
    return 0;// Завершить симуляцию
}
```

Результаты моделирования

```
@1 ns Starting test
@6 ns Triggering e1
@6 ns Time before wait 3 ns
@9 ns Time after wait 3 ns
@11 ns Got Trigger e1
@31 ns Triggering e2
@32 ns Got Trigger e2
@51 ns Done waiting for 20ns or event e1
Terminating Simulation
```

3.14.2. Next_trigger ()

`Next_trigger()` используется с методами процесса, которые не являются потоками. Функция `next_trigger` не приостанавливает экземпляр процесса метода: процесс метода не может быть приостановлен, но всегда выполняется до завершения, прежде чем вернуть управление ядру. Этим он отличается от метода `wait ()`, который используется с потоками.

`Next_trigger ()`: при отсутствии статической чувствительности для этого конкретного экземпляра процесса этот процесс не должен снова включаться во время текущего моделирования.

`Next_trigger (event)`: процесс должен запускаться, когда о событии, переданном в качестве аргумента, получено уведомление.

`Next_trigger (double, sc_time_unit)`: процесс должен запускаться по истечении заданного времени.

`Next_trigger (double, sc_time_unit, event):` процесс должен быть инициирован, когда получено уведомление о событии или по истечении заданного времени, которое происходит первым.

Листинг 3.21

Программа sc_next_trigger

```
#include <systemc.h>

SC_MODULE (next_trigger_example) {
  sc_in<bool> clock;

  sc_event  e1,e2;
  int cnt;

  void do_test1() {
    switch (cnt) {
      case 0 : cout << "@" << sc_time_stamp() <<"
Default trigger clk triggered"<<endl;
               next_trigger(e1);
               break;
      case 1 : cout << "@" << sc_time_stamp() <<"
Event e1 triggered"<<endl;
               next_trigger(10, SC_NS);
               break;
      case 2 : cout << "@" << sc_time_stamp() <<"
Event e1 occured or time 10ns passed"<<endl;
               next_trigger(e1 | e2);
               break;
      case 3 : cout << "@" << sc_time_stamp() <<"
Event e1 or e2 triggered"<<endl;
               break;
      default :cout << "@" << sc_time_stamp() <<"
Default trigger clk triggered"<<endl;
               break;
    }
    cnt ++;
  }

  void do_test2() {
    while (true) {
      wait(2);
      // Переключение событием e1
```

```
          cout << "@" << sc_time_stamp() <<"
Triggering e1"<<endl;
          e1.notify();
          wait(20);
          cout << "@" << sc_time_stamp() <<"
Triggering e2"<<endl;
          e2.notify();
          // Ждать 2 положительных фронта тактов
          wait(2);
          cout << "@" << sc_time_stamp() <<"
Terminating simulation"<<endl;
          sc_stop(); /* sc_stop запускает завершение
моделирования*/
        }
      }

      SC_CTOR(next_trigger_example) {
        cnt = 0;
        SC_METHOD(do_test1);
          sensitive << clock;
        SC_CTHREAD(do_test2,clock.pos());
      }
    };

    int sc_main (int argc, char* argv[]) {
      sc_clock clock ("my_clock",1,0.5);

      next_trigger_example  object("wait");
        object.clock (clock);

      sc_start(0); /* Первый раз вызывается
планировщик инициализации */
    sc_start();  /* Запустить симуляцию до появления
sc_stop */
    return 0;// Завершение симуляции
    }
```

Результаты моделирования

```
@0 s Default trigger clk triggered
@2 ns Triggering e1
@2 ns Event e1 triggered
@12 ns Event e1 occured or time 10ns passed
@22 ns Triggering e2
@22 ns Event e1 or e2 triggered
@22500 ps Default trigger clk triggered
@23 ns Default trigger clk triggered
@23500 ps Default trigger clk triggered
@24 ns Default trigger clk triggered
@24 ns Terminating simulation

Info: /OSCI/SystemC: Simulation stopped by user.
```

3.15. Методы

Как упоминалось ранее, SystemC имеет более чем один тип процесса. `SC_METHOD` процесс более простой, чем `SC_THREAD`. Однако, эта простота делает его более трудным для некоторых стилей моделирования.

Сходство наименования между процессом SC_METHOD и обычным объектно-ориентированного методом передает свое название. Он выполняется без прерывания и возвращается к вызывающему (планировщику). В противоположность этому, процесс SC_THREAD является более родственным отдельной операционной системе потоков с возможностью прерывания и возобновления. Переменные, выделенные в процессах SC_THREAD, являются неизменными. SC_METHOD процессы должны объявлять и инициализировать переменные каждый раз, когда метод вызывается. По этой причине процессы SC_METHOD обычно используют модуль локальных элементов данных, объявленых в SC_MODULE. SC_THREAD процессы как правило, используют локально объявленные переменные.

3.15.1. Методы sc_start() и sc_stop()

Метод `sc_start()` запускает фазу моделирования, которая состоит из инициализации и выполнения. Метод может принимать параметр типа sc_time, который является ограничением максимального времени моделирования. Без параметра `sc_start()` запускает фазу моделирования, которая будет протекать бесконечно, пока в программе не встретится метод `sc_stop()`, который принудительно завершает моделирование.

3.15.2. Метод wait()

Метод `wait()` используется в SystemC для того чтобы моделировать задержки реальных действий (например: механических воздействий, химических реакций или распространения сигнала). С

помощью метода можно приостановить выполнение процесса SC_THREAD на промежуток времени или до появления какого-либо события:

```
sc_time t_delay(25, SC_FS);wait(t_delay);
```

3.15.3. Метод sc_time_stamp()

Данный метод позволяет определить в момент вызова, сколько времени прошло с начала моделирования, например:

```
sc_time t_delay1(10, SC_NS);
sc_time t_delay2(25, SC_NS);
cout << "Текущее время моделирования: " <<
sc_time_stamp();
wait(t_dalay1);
cout << "Текущее время моделирования: " <<
sc_time_stamp();
 wait(t_delay2);
cout << "Текущее время моделирования: " <<
sc_time_stamp();
```

В результате исполнения данного проекта, мы получаем на экране следующие сообщения:

Текущее время моделирования: 0 s.
Текущее время моделирования: 10 ns.
Текущее время моделирования: 35 ns

3.16. Динамическая чувствительность для SC_METHOD: next_trigger ()

SC_METHOD процессы позволяют динамически определять их чувствительность с помощью метода next_trigger (). Этот метод имеет тот же синтаксис, что и wait() метод, но с несколько иным поведением.

```
next_trigger(time);
next_trigger(event);
next_trigger ;// любой из этих
next_trigger ;//все эти обязательны
next_trigger(timeout, event); /* событие с
таймаутом*/
next_trigger(timeout, ;// любой + таймаут
next_trigger(timeout, ; //все и таймаут
next_trigger() ; /* восстановить статическую
чувствительность*/
```

Как и в случае ожидания, несколько синтаксисов событий не уточняют порядок событий. Так, с `next_trigger (evt1 & evt2)`, не представляется возможным узнать, какое событие произошло в первую очередь. Можно лишь утверждать, что оба `evt1` и `evt2` случилось.

Метод `wait` приостанавливает процессы `SC_THREAD`. Однако, процессы `SC_METHOD` не могут приостанавливаться. Метод `next_trigger` эффективен для временной настройки списка чувствительности, который влияет на `SC_METHOD`. Метод `next_trigger` можно вызывать несколько раз, и каждый новый вызов переопределяет предыдущие. Последний выполненный `next_trigger` перед возвращением из процесса определяет чувствительность для повторного вызова процесса. Вызов инициализации имеет жизненно важное значение для выполнения работы.

Следует отметить, что для каждого пути через `SC_METHOD` необходимо указать по крайней мере один `next_trigger` для того, чтобы процесс был вызван планировщиком. Без создание `next_trigger` или статической чувствительности `SC_METHOD` никогда не будет выполняться снова. Предостережение может относится к размещению `next_trigger` по умолчанию в качестве первой команды `SC_METHOD`

, так как последующие `next_triggers` будут переписывать все предыдущие. Лучший способ справиться с этой проблемой существует и состоит в следующем.

3.17. Статическая чувствительность для процессов

Мы обсудили технику динамического (т.е. во время моделирование), определения, как возобновится процесс (либо как `SC_THREAD`, используя ожидание, или с помощью `SC_METHOD`, используя `next_trigger`). SystemC обеспечивает другой тип чувствительности для процессов, который называется статической чувствительностью. Статическая чувствительность устанавливает параметры для возобновления в процессе разработки (то есть, перед началом моделирования). После создания статические параметры чувствительности не могут быть изменены (т.е. они статические). Как мы увидим, можно переопределить статическую чувствительность,.

Статическая чувствительность устанавливается с помощью вызова чувствительного метода `sensitive()` или перезагрузки оператора потока `operator <<`, который помещен сразу после регистрация процесса. Статическая чувствительность применяет только самую последнюю регистрацию процесса. Чувствительность `sensetive` может быть указана несколько раз. Есть два стили синтаксиса:

```
    /* ВАЖНО: Должен следовать за регистрацией
процесса*/
    sensitive << event [<< event] ;// потоковый стиль
    sensitive (event [, event] ); /* функциональный
стиль*/
```

Мы предпочитаем стиль потоковой передачи, так как он чувствует себя более объектно-ориентированным, уменьшает набор операторов и держит нас сосредоточеными на C++.

Иногда возникает необходимость уточнить некоторые процессы, которые не следует инициализировать. Эту ситуацию в SystemC обеспечивает метод `dont_initialize`. Синтаксис имеет следующий вид:

```
    /* ВАЖНО: Должен следовать за регистрацией
процесса*/
    dont_initialize();
```

Использование `dont_initialize` требует списка статической чувствительности. В противном случае, не было бы ничего, чтобы начать процесс. Теперь наш модуль содержит:

```
    SC_METHOD(attendant_method);
    sensitive(event [, event);
    dont_initialize() ;
```

В свете ограничения, что `sc_events` может иметь только одно невыполненное расписание, `sc_event_queues` были добавлены в SystemC версия 2.1. Эти дополнения позволяют одно событие запланировать несколько раз даже в одно и то же время. Когда события запланированы на то же время, каждое происходит в отдельном дельта - цикле.

`sc_event_queue` немного отличается от `sc_event`. Во-первых, `sc_event_queue` объекты не поддерживают немедленное уведомление и, очевидно, нет необходимости создавать их очередь. Во-вторых, метод `.cancel ()` заменяется `.cancel_all ()`, чтобы подчеркнуть, что он отменяет все внешние `sc_event_queue` уведомления.

```
    sc_event_queue action;
    sc_time now(sc_time_stamp()); /* соблюдать текущее
время */
    action.notify(20, SC_MS);// расписание на 20 мс
    action.notify(1.5,SC_NS);/* другой в течение 1,5
нс с этого момента */
```

```
    action.notify(1.5,SC_NS);/* другое идентичное
действие */
    action.notify(3.0,SC_NS);/* другой на 3.0 нс с
этого момента */
    action.notify(SC_ZERO_TIME);/* для следующего
дельта-цикла */
    action.notify(1,SC_SEC);// в течение 1 секунды
    action.cancel_all(); /* полностью отменить все
действия */
```

Метод `.cancel ()` в настоящее время не реализуется; хотя, очевидно, расширение может быть использовано, чтобы отменить уведомления в определенное время. Еще одним расширением может быть получение информации о том, сколько невыполненных уведомлений существует `(.pending ())`

3.18. Типы данных и операторы

В таблице 3.1 представлены типы данных, поддерживаемые SystemC. В таблице 3.2 представлены операции над данными, поддерживаемые SystemC.

Таблица 3.1

Основные типы данных, поддерживаемые SystemC

Тип данных	Описание
sc_bit	Одиночный бит, принимающий значение true или false. Использовать данных тип не рекомендуется, более предпочтительно применение типа bool
sc_bv< n>	Вектор, содержащий n бит. Рекомендуется использовать sc_uint <n>, где это возможно
sc_logic	Одиночный бит, который принимает значения 0, 1, X, Z
sc_lv<n>	Вектор, содержащий n бит, типа sc_logic
sc_int<n>	Вектор, содержащий n целых чисел, размером 64 бит
sc_uint< n>	Беззнаковое sc_int <n>
sc_bigint<n>	Вектор, содержащий n целых чисел, размером более 64 бит
sc_biguint< n>	Беззнаковое sc_bigint <n>

К поддерживаемым типам данных также относятся другие типы данных C++, такие как `bool, int, unsigned int, long, unsigned long, char, unsigned char, short, unsigned short, struct, enum.`

Таблица 3.2.

Операции над данными, поддерживаемые SystemC

Операции	Описание
&(and), |(or), ^(xor), and ~(not)	Логические операции
<<(shift left) and >>(shift right)	Логические сдвиги
=, &=, |=, ^=, +=, -=, *=, /=, and %=	Префиксные операции
==, !=	Равенство
<, <=, >, >=	Сравнение
++ и --	Инкремент и декремент
[x]	Индексирование
(x-y)	Беззнаковое sc_bigint <n>
(x,y)	Конкатенация
to_uint( ) и to_int( )	Преобразование типов

Перечисленные операции над данными в основном совпадают с использоваанными в C++.

3.18.1. Тип sc_bit

Тип `sc_bit` представляет собой двухзначный тип данных, представляющий один бит. Переменная типа `sc_bit` может иметь только значение '0' (ложь) или '1' (истина). Этот тип полезен для моделирования части конструкции, где Z (импеданс) или X (неизвестные) значения не являются необходимыми.

Есть целый ряд логических операторов и операторов сравнения, которые работают с `sc_bit` в том числе:

```
Побитовые &(and) |(or) ^(xor) ~(not)
Присваивание = &= |= ^=
Сравнение  == !=
```

Значения присваиваются с помощью символьных литералов '1' и '0'. При выполнении булевых операций объекты `sc_bit` типа могут быть смешаны с C/C++ типами BOOL. Объекты типа `sc_bit` хороши для представления отдельных битов конструкции, где будут выполнены логические операции. Чтобы объявить объект типа `sc_bit` используется следующий синтаксис:

```
sc_bit s;
```

Тип `sc_bit` может содержать два значения, поэтому его нельзя использовать для моделирования реального оборудования. В реальном оборудовании мы имеем «0», «1», «X» и «Z».

Пример программы с типом `sc_bit`.

Листинг 3.22

Программа SC_BIT

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  // Объявление sc_bit
  sc_bit             enable;
  sc_bit             read_en;
  // Назначить значение sc_bit
  enable = '1';
  cout <<"Value of enable  : " << enable << endl;
  // Назначить sc_bit другой sc_bit
  read_en = enable;
  cout <<"Value of read_en : " << read_en <<
endl;

  return 1;
}
```

Результат моделирования:

```
Value of enable  : 1
Value of read_en : 1
```

3.18.2. Тип sc_logic

Более общим одинобитным типом является тип `sc_logic`. Этот тип имеет 4 значения: '0' (ложь), '1' (истина), 'X' (неизвестно), и 'Z' (высокое сопротивление или плавающее). Этот тип может быть использован, чтобы моделировать конструкции с несколькими шинами драйвера передачи X, значений запуска, и плавающими шинами. Тип `sc_logic` имеет наиболее распространенные значения, используемые в VHDL и Verilog моделировании на уровне RTL. Тип `sc_logic` имеет ряд логических операторов, сравнения и присваивания, которые могут быть использованы с объектами этого типа. К ним относятся следующие:

Операторы sc_logic

```
Битовые &(and) |(or) ^(xor) ~(not)
Присвоение = &= |= ^=
Сравнение == !=
```

Эти операторы реализованы таким образом, что операнды типа `sc_logic` могут быть смешаны с операндами типа `sc_bit`. Один из операндов должен быть тип `sc_logic`, другие операнды могут быть `sc_logic` или `sc_bit`.

Значения присваиваются объектам `sc_logic` с использованием литералов символов, показанных ниже:

- '0' - 0 или ложное значение
- '1' - 1 или истинное значение
- 'X' - неизвестная или неопределенная величина;
- 'Z' - высокое сопротивление или плавающее значение.

Пример присваивания показан ниже:

```
sc_logic x; // объявление объекта
x = '1'; // присвойте значение 1
x = 'Z'; // присвойте значение Z
```

Операторы сравнения == и ! = реализованы так, что дизайнер может сравнить два `sc_logic` объекта, `sc_logic` объект и объект `sc_bit`, или `sc_logic` объект с один из значений литерала. Следующие сравнения могут быть реализованы:

```
sc_bit x;
sc_logic y,z;
x == y; // sc_bit и sc_logic
y != z; // sc_logic и sc_logic
y == '1' // sc_logic и символьный литерал
```

Оператор присваивания позволяет присвоить значение литерала или другого sc_logic объекта к объекту sc_logic. Дополнительно sc_bit может быть преобразован в sc_logic посредством назначения. Следующие присваивания являются преобразованиями:

```
sc_bit x;
sc_logic y;
x = y; // преобразование sc_logic в sc_bit
y = x; // проебразование sc_bit в sc_logic
```

Первое задание будет преобразовать тип sc_logic к типу sc_bit. Так как sc_bit объект имеет 2 значения в то время как sc_logic тип имеет 4 значения, значения 'Z' и 'X' не могут быть преобразованы в sc_bit. Если значение sc_logic объекта 'Z' или 'X', когда происходит назначение, результат выполнения задания не определен и во время выполнения будет выпущено предупреждение.

Рассмотрим ещё два примера использования (листинг 3.23).

Листинг 3.23

Программа SC_LOGIC

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  // Объявление sc_logic
  sc_logic       read_en;
  sc_logic       pad;
  sc_logic       enable;
  sc_bit         no_x_z;
  // Назначить значения
  pad = 'z';
  cout <<"Value of pad     : " << pad << endl;
  enable = '0';
  cout <<"Value of enable  : " << enable << endl;
  read_en = ~enable;
  cout <<"Value of read_en : " << read_en <<
endl;
  // Логическая операция
  if (pad == '1') {
    cout <<"Pad is 1"<< endl;
  } else {
    cout <<"Pad is not 1"<< endl;
  }
  // Назначить тип sc_bit
  no_x_z = enable; // Assign 0/1 value
```

```
    cout <<"Value of no_x_z  : " << no_x_z  <<
endl;
    no_x_z = pad; // Назначить Z
    cout <<"Value of no_x_z  : " << no_x_z  <<
endl;
    // Назначить отброшенные значения
    pad = sc_logic ('x');
    cout <<"Value of pad     : " << pad << endl;

    return 1;
  }
```

Результаты моделирования:

```
Value of pad     : Z
Value of enable  : 0
Value of read_en : 1
Pad is not 1
Value of no_x_z  : 0

Warning: (W211) sc_logic value 'Z' cannot be converted to bool
In file: ../../../../src/sysc/datatypes/bit/sc_logic.cpp:91
Value of no_x_z  : 1
Value of pad     : X
```

Типы данных `sc_bit` и `sc_logic` могут использоваться в сравнении между собой и с литералами (листинг 3.24).

Листинг 3.24

Программа SC_LOGIC_COMP

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  // Объявить sc_bit
  sc_bit x;
  // Объявить sc_logic
  sc_logic y,z;
  // Назначить значения для x,y,x
  x = '1';
  y = '1';
  z = 'x';
  // sc_bit и sc_logic
  if (x == y) {
    cout <<"x = y " << endl;
  } else {
    x = y; // Назначить sc_logic для sc_bit
    cout <<"x != y " << endl;
```

```
    cout<<"New Value of x : "<< x << endl;
  }
  // sc_logic and sc_logic
  if (y != z) {
    y = z; // Назначить sc_bit для sc_logic
    cout <<"y != z " << endl;
    cout<<"New Value of y : "<< y << endl;
  }
  // sc_logic и символьный литерал
  if (y == 'x') {
    cout<<"y equal to value 'X'"<< endl;
  }

  return 1;
}
```

Результат моделирования:

```
Value of pad     : Z
Value of enable  : 0
Value of read_en : 1
Pad is not 1
Value of no_x_z  : 0
```

3.18.3. Битовый вектор произвольной длины sc_bv

SystemC также содержит двухзначный вектор произвольной длины, который будет использоваться для больших манипуляций с `bit_vector`.

Тип `sc_bv` вводит некоторые новые операторы, которые выполняют сокращение передачи битов. Эти операторы принимают весь набор битов операнда и генерируют один бит результата. Например, чтобы выяснить, имеет ли шина данных все 0, может быть выполнена следующая операция:

```
sc_bv<64> databus;
sc_logic result;
result = databus.or_reduce();
```

Если шина данных содержит одну или более 1, значения результата будет равен 1. Если нет единиц, значения результата будет 0 и указывает, что шина данных имеет все 0 .

`Sc_bit` является одним битом. SystemC предоставляет `sc_bv` или SystemC битовый вектор для многобитового векторного объявления. Ширина вектора указана в целых числах. Крайний правый индекс вектора равен 0 и также является наименее значимым битом. Ширина M задает размер и направление вектора от (M-1) до 0, причем бит M-ный является самым значимым битом.

Если присваивание переменной битового вектора меньше размера (ширины), то остальные биты расширяются с 0. Если они больше, то они усекаются.

Тип sc_bv вводит некоторые новые операторы и методы, которые выполняют сокращение битов, выбор диапазона. Сокращения битов и выбор диапазона не могут быть выполнены в порту и в сигналах. Для sc_lv типов не допускаются арифметические операции, вместо этого они должны выполняться в sc_int или sc_uint и затем назначаться в sc_lv.

Листинг 3..25

Программа SC_BV

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  sc_bv<8>  data_bus  ;
  sc_bv<16> addr_bus  ;
  sc_bit    parity    ;
  // Назначить значение для sc_bv
  data_bus = "00001011";
  cout <<"Value of data_bus : " << data_bus <<
endl;
  // Использовать оператор диапазона
  addr_bus.range(7,0) = data_bus;
  cout <<"Value of addr_bus : " << addr_bus <<
endl;
  /* Назначить обратную ссылку на шину адреса,
используя оператор диапазона*/
  addr_bus.range(0,7) = data_bus;
  cout <<"Value of addr_bus : " << addr_bus <<
endl;
  // Используйте бит для установки значения
  addr_bus[10] = "1";
  cout <<"Value of addr_bus : " << addr_bus <<
endl;
  // Использовать оператор сокращения
  parity  = data_bus.xor_reduce();
  cout <<"Value of parity   : " << parity <<
endl;

  return 1;
}
```

Результат моделирования:

```
Value of data_bus : 00001011
Value of addr_bus : 0000000000001011
Value of addr_bus : 0000000011010000
Value of addr_bus : 0000010011010000
Value of parity   : 1
```

3.18.4. Беззнаковые и знаковые целые с фиксированной точностью sc_int <n> и sc_uint <n>

Некоторые системы требуют арифметические операции над фиксированным размером арифметических операндов. Знаковые и беззнаковые целые типы с фиксированной точностью обеспечивают эту функциональность в SystemC. В C++ тип INT зависит от машины, но обычно 32 бит. Если дизайнер собирался использовать только 32 битные арифметические операции, то этот тип будет работать. Однако целый тип в SystemC предоставляет целые числа от 1 до 64 бит в знаковых и беззнаковых формах.

Лежащий в основе реализация фиксированный тип точности представляет собой 64-битное целое. Все операции выполняют с 64 битовыми целыми числами, а затем превращают в соответствующий размер через усечение результата. Если дизайнер умножает два 44 разрядных целых числа, максимальный размер результата составляет 64 бит, поэтому только 64 бита сохраняются. Если требуется результат в 44 бита, то 20 бит удаляются. Если требуется большая точность необходимо использовать целые с произвольной точностью.

Самая быстрая скорость моделирования будет получена с помощью встроенных в C++ типов данных `int`, `long` и т.д. Однако, эти типы работают только для фиксированного размера данных 8, 16 или 32 бита. Второй способ быстрого моделирования может быть реализован с помощью целых с фиксированной точностью.

Самое медленное время моделирования будет в случае использования типа `int` с произвольной точностью. Так всякий раз, когда это возможно следует использовать тип `int` с фиксированной точностью вместо целых с произвольной точностью для максимальной скорости моделирования.

Тип `sc_int <n>` является целым числом с фиксированной точностью, а тип `sc_uint <n>` является целым числом с фиксированной точностью без знака. Основополагающие операции используют 64 бита, но размер результата определяется при объявлении объекта. Например, следующее объявление объявляет 64 битное целое число без знака и 48-битное целое число без знака.

```
sc_uint<64> x;
```

```
sc_uint<48> y;
```

Целые типы имеют очень богатый набор операторов, которые работают с ними, как показано в списке ниже:

Битовые (Bitwise) ~ & | ^ >> <<

Арифметические (Arithmetic) + - * / %

Присваивание (Assignment) = += -= *= /= %= &= |= ^=

Эквивалентность (Equality) == !=

Отношения сравнения (Relational) < <= > >=

Автоинкремент (Autoincrement) ++

Автодекремент (Autodecrement) --

Выбор бита (Bit Select) [x]

Выбор часть (Part Select range())

Контатенация (Concatenation(,))

Битовые операторы работают на операнды бит за битом. Оператор (~) не инвертирует все биты, и операторы сдвига будут выполнять сдвиг влево (<<) или вправо (>>) на заданное количество битов. Пример показан ниже:

```
sc_int<16> x, y, z;
z = x & y; /* выполнить и работать на бит x и y
по битам*/
z = x >> 4; /* присвойте x сдвинутым вправо на 4
бита на z*/
```

С добавлением арифметических операторов для целых типов SystemC, новые операторы присваивания также доступны. Например оператор + = позволит более лаконично описать следующего утверждения:

```
x = x + y; // традиционный путь
x += y; // краткий метод
```

Чтобы выбрать один бит целого числа, используйте оператор выбора бита, как показано ниже:

```
sc_logic mybit;
sc_uint<8> myint;
mybit = myint[7];
```

Чтобы выбрать более одного бита, используется метод диапазона, как показано ниже:

```
sc_uint<4> myrange;
sc_uint<32> myint;
myrange = myint.range(7,4);
```

Наконец оператор конкатенации может быть использован, чтобы сделать более высокое значение из одного или более меньших значений. Пример показан ниже:

```
sc_uint<4> inta;
sc_uint<4> intb;
sc_uint<8> intc;
intc = (inta, intb);
```

Операнды `inta` и `intb` объединяются вместе, чтобы сформировать 8 битное целое число, которое затем присваивается целому `intc`.

Автоинкремент и автодекремент - еще один способ сделать описание более кратким и лаконичным. Оператор автоматического приращения будет увеличивать операнд, прикрепленный к оператору, а оператор автоматического декремента будет уменьшать операнд. Например, вместо написания:

```
a = a + 1;
```

оператор автоинкремента позволяет записать:

```
a++;
```

Переменная типа `sc_uint` (без знака) может быть преобразована в тип `sc_int` (со знаком) оперетором присваивания = . Таким же образом, переменные типа `sc_int` могут быть преобразованы к `sc_uint`. Когда оператор = используется, любые дополнительные биты удаляются, биты знака добавляются и могут быть расширены по мере необходимости. Пример показан ниже:

```
sc_uint<8> uint1, uint2;
sc_int<16> int1, int2;
uint1 = int2; // конвертировать int в uint
int1 = uint2; // конвертировать uint в int
```

В первой записи целое число преобразуется в целое число без знака. Абсолютное значение `int2` будет присвоено `uint1`. Если `int2` имеет отрицательное значение, только величина будет назначена `uint1`. Так как `int2` составляет 16 бит, в то время как `uint1` 8 бит, `uint2` будет преобразовано в число без знака до 64 бит, а затем обрезается до 8 бит перед тем присвоением `uint1`.

Во второй команде `uint2` присваивается `int1.` Сначала `uint2` будет преобразован в 64-битное значение, а затем усечен и присвоен `int1`.

Тип `sc_int` и `sc_uint` могут использоваться с C++ целочисленными типами без ограничений. C++ целые типы могут быть свободно смешаны с типами SystemC.

Листинг 3.26

Программа SC_INT

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  sc_int<1>  bit_size    = 0;
  sc_int<4>  nibble_size = 1;
  sc_int<8>  byte_size   = 2;
  sc_int<32> dword_size  = 3;
  /*sc_int<128> addr; sc_int не может быть больше
64*/
  // Выполнить автоматическое увеличение
  dword_size ++;
  cout <<"Value of dword_size : " << dword_size
<< endl;
  // Краткий метод прибавления
  byte_size += nibble_size;
  cout <<"Value of byte_size  : " << byte_size <<
endl;
  // Выбор бита
  bit_size = dword_size[2];
  cout <<"Value of bit_size   : " << bit_size <<
endl;
    // Выбор диапазона
      nibble_size = dword_size.range(4,1);
    //Невозможно назначить вне диапазона
  cout <<"Value of nibble_size: " << nibble_size
<< endl;
  // Соединение (контатенация)
  dword_size =
(byte_size,byte_size,byte_size,byte_size);
  cout <<"Value of dword_size : " << dword_size
<< endl;

  return 1;
}
```

Результаты моделирования:

```
Value of dword_size : 4
Value of byte_size  : 3
Value of bit_size   : -1
Value of nibble_size: 2
Value of dword_size : 50529027
```

Sc_uint - целое число без знака, и оно является фиксированным целым числом точности размера 64 бита. В базовых операциях используются 64 бита, но размер результата определяется при объявлении объекта. Операнды типа sc_uint могут быть преобразованы в тип sc_int и наоборот с помощью операторов присваивания. При назначении целого числа беззнаковому операнду целочисленное значение в форме дополнения 2 интерпретируется как беззнаковое число. При назначении беззнакового для знакового операнда выполняется беззнаковое расширение до 64-битного беззнакового числа и затем усекается, чтобы получить знаковое значение.

Sc_uint имеет богатые операторы, как в случае с sc_int. Посмотрите пример с sc_uint для детализации.

Листинг 3.27

Программа SC_UINT

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  sc_uint<1>  bit_size    = 0;
  sc_uint<4>  nibble_size = 1;
  sc_uint<8>  byte_size   = 2;
  sc_uint<32> dword_size  = 3;
  /*sc_int<128> addr; sc_int не может быть больше
64 */
// Выполнить автоматическое увеличение
  dword_size ++;
  cout <<"Value of dword_size : " << dword_size
<< endl;
  // Краткий метод сложения
  byte_size += nibble_size;
  cout <<"Value of byte_size  : " << byte_size <<
endl;
  // Выбор бита
  bit_size = dword_size[2];
  cout <<"Value of bit_size   : " << bit_size <<
endl;
  // Выбор диапазона
  nibble_size = dword_size.range(4,1); // Can not
assign out of range
  cout <<"Value of nibble_size: " << nibble_size
<< endl;
```

```
      // Соединенные
      dword_size =
(byte_size,byte_size,byte_size,byte_size);
      cout <<"Value of dword_size : " << dword_size
<< endl;

      return 1;
    }
```

Результаты моделирования:

```
Value of dword_size : 4
Value of byte_size  : 3
Value of bit_size   : 1
Value of nibble_size: 2
Value of dword_size : 50529027
```

3.18.5. Знаковые и беззнаковые типы целых с произвольной точностью sc_bigint и sc_biguint

Тип `sc_bigint` является двойным дополнением целого числа со знаком любого размера. Тип `sc_biguint` является целым числом без знака любого размера. При использовании целых чисел произвольной точности точность, используемая для расчетов, зависит от размеров используемых операндов,. Посмотрите на пример, приведенный ниже:

```
sc_biguint<128> b1;
sc_biguint<64> b2;
sc_biguint<150> b3;
b3 = b1*b2;
```

В этом примере `b1`, 128-битное целое число умножается на `b2`, 64 битное целое число. Результат будет 192-битное целое. Тем не менее, поскольку `b3` составляет всего 150 бит шириной 42 бита удаляются из результата перед назначением на `b3`.

Иногда операнды должны быть больше 64 бит. Для этих типов проектов `sc_int` работать не будет. Для этих случаев SystemC предоставляет тип `sc_bigint` (произвольное целое число со знаком). Эти типы позволяют конструктору работать с целыми числами любого размера, ограниченными только системными ограничениями. Арифметические операторы и другие операторы также используют произвольную точность при выполнении операций. SystemC определяет `MAX_NBITS` в `sc_constants.h`, чтобы ограничить максимальный размер битов для `sc_bigint` до 512. Тип `sc_bigint` представляет

собой целое число со знаком, состоящее из двух цифр любого размера. Sc_bigint имеет много операторов, как в случае с sc_int. Посмотрите пример для sc_bigint.

Листинг 3.28

Программа SC_BIGINT

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  sc_bigint<128> large_size ;
  // Оператор сдвига
  large_size =  1000 << 1;
  cout <<"Value of large_size : " << large_size
<< endl;

  return 1;
}
```

Результаты моделирования:

```
Value of large_size : 2000
```

Sc_biguint - это то же самое, что и sc_bigint, только отличие - sc_biguint без знака. Эти типы позволяют конструктору работать с целыми числами любого размера, ограниченными только системными ограничениями. Арифметические операторы и другие операторы также используют произвольную точность при выполнении операций. SystemC определяет MAX_NBITS в sc_constants.h, чтобы ограничить максимальный размер битов для sc_biguint до 512. Sc_biguint имеет те же операторы, как в случае с sc_int. Посмотрите пример для sc_biguint.

Листинг 3.29

Программа SC_BIGUINT

```
#include <systemc.h>

int sc_main (int argc, char* argv[]) {
  sc_biguint<128> a  = 30000;
  sc_biguint<128> b  = 20000;
  sc_biguint<256> c  = 0;
  // оператор умножения
  c = a * b;
  cout <<"Value of c : " << c << endl;

  return 1;
```

```
}
```

Результаты моделирования

```
Value of c : 600000000
```

3.18.6. Логический вектор произвольной длины sc_lv <N>

Для частей конструкции, которые должны быть смоделированы с возможностями трехуровневых значений и содержат еще элементы, которые шире, чем 1 бит, SystemC содержит тип, который называется логический вектор `sc_lv <N>.` Этот тип представляет собой произвольное значение вектора длины, где каждый бит может иметь одно из четырех значений. Эти значения являются точно такими же, как и четыре значения типа `sc_logic`. Тип `sc_lv <n>` на самом деле просто массив объектов переменной размера `sc_logic`.

Для объявления сигнала типа `sc_lv<n>` используют следующий синтаксис:

```
sc_signal<sc_lv<64> > databus; /* дополнительное пространство обязательно*/
```

Эта декларация описывает широкий 64 битный сигнал под названием шина данных, в котором каждый из битов сигнала может иметь значение '0', '1', 'X' и 'Z'. Этот сигнал может управлять рядом источников для моделирования шины с тремя состояниями.

Очень важно отметить, что требуется дополнительное пространство после первого знака >, чтобы позволить компиляцию декларации.

Для преобразования типа `sc_lv` к арифметическому типу надо использовать оператор =. Это показано ниже:

```
sc_uint<16> uint16;
sc_int<16> int16;
sc_lv<16> lv16;
lv16= uint16; // конвертировать uint в lv
int16 = lv16; // конвертировать lv to int
```

Первый оператор преобразует целое число без знака в логический вектора 16 бит. Второй оператор преобразует логический вектор к 16 битному целому числу. Любые знаки X или Z в логическом векторе будут выдавать предупреждение во время выполнения, и результаты будут неопределенными.

Общая функция необходима, чтобы правильно моделировать шины с тремя состояниями и появилась возможность выключить все драйвера к шине. Для выполнения этого шага надо присвоить строку значений 'Z' к объекту `sc_lv`.. Это показано ниже:

```
sc_lv<16> bus1;
if (enable) {
```

```
bus1 = in1
} else {
bus1 = "ZZZZZZZZZZZZZZZZ";
}
```

Это назначение будет присваивать значение Z для всех 16 мест BUS1. Строка символов может содержать любую комбинацию из четырех значений, '0', '1', 'X', и 'Z'. Так другая правомерная строка для BUS1 будет следующей:

```
bus1 = "01XZ01XZ01XZ01XZ";
```

Чтобы напечатать удобочитаемую строку символов значения из объекта `sc_lv`, используют метод `to_string ()`, как показано:

```
sc_lv<32> bus2;
cout << "bus = " << bus2.to_string();
```

3.19. Оператор сравнения

Для скалярных типов встроенные операторы сравнения используются для определения того, не изменилось ли значение, которое генерируют события. Для определенных пользователем типов, таких как `packet_type`, используются в симплексном примере, когда надо предоставить == оператор. Рассматривая `packet.h` ,мы видим следующее:

```
inline bool operator == (const packet_type& rhs)
const
{
return (rhs.info == info && rhs.seq == seq &&
rhs.retry == retry);
}
```

Этот метод определяет поля, которые должны быть сравнены и как сделать сравнение. Событие происходит, если результат сравнения указывает на то, что предыдущий пакет и новый пакет различны.

3.20. Трассировка определенного пользователем типа

Обратите внимание на то, что файл `packet.h` также отслеживает сигналы заданного пользователем типа `packet_type`. Поскольку этот тип имеет ряд полей, необходимо указать отслеживание каждого поля, чтобы увидеть содержимое пакета. Этот процесс не является автоматическим. Вы можете определить специальный метод трассировки пользователя, который вызывается, когда объект этого типа прослеживается. Этот метод пользователя может быть определен в определенных пользователем типах.

Оглядываясь назад на файлы `packet.h` и `packet.cc`, мы видим, что определяемый пользователем тип `packet_type` имеет метод

sc_trace, определенный в packet.h. Эта метод определяет как отследить объект типа packet_type. Декларация о методе, типы аргументов и возвращаемого значения находятся в файле packet.h, как показано ниже:

```
    extern
    void sc_trace(sc_trace_file *tf, const
packet_type& v,
    const sc_string& NAME);
```

Обратите внимание на то, что второй аргумент имеет тип packet_type, что делает этот метод уникальным. Файл packet.cc содержит реализацию метода sc_trace как показано ниже:

```
    void sc_trace(sc_trace_file *tf, const
packet_type& v,
    const sc_string& NAME) {
    sc_trace(tf,v.info, NAME + ".info");
    sc_trace(tf,v.seq, NAME + ".seq");
    sc_trace(tf,v.retry, NAME + ".retry");
    }
```

Реализация метода трассировки имеет след (график) для каждого поля структуры. Эта метод вызывается конструктором, чтобы выполнить трассировку по сигналу типа packet_type, и автоматически создается компилятором. Каждый вызов метода трассировки будет выполнять след на всех полях определенного пользователем типа.

3.22. Типы с фиксированной точкой

Когда конструкторы создают модели на высоком уровне, числа с плавающей точкой полезны для моделирования арифметических операций. Числа с плавающей точкой могут обрабатывать очень большой диапазон значений и легко масштабируются. В аппаратных средствах типы данных с плавающей точкой, как правило, преобразуются или встраиваются в качестве типов данных с фиксированной точкой, чтобы минимизировать количество оборудования, необходимого для реализации функциональности. Для моделирования поведения аппаратных средств с фиксированной точкой дизайнеры должны использовать битовые точные типы данных с фиксированной точкой. Типы с фиксированной точкой также используются для разработки программного обеспечения DSP (Digital Signal Processor).

SystemC содержит знаковые и беззнаковые типы данных с фиксированной точкой, которые могут быть использованы для точной аппаратной модели. В SystemC фиксированные типы данных точек являются точными до уровня битов и поддерживают ряд функций, которые позволяют выполнить высокий уровень моделирования. Эти функции

включают в себя моделирование квантования и поведение переполнения при высоком уровене.

Есть 4 основных типа, используемые для моделирования типов с фиксированной точкой в SystemC. Это:

- `sc_fixed`
- `sc_ufixed`
- `sc_fix`
- `sc_ufix`

Типы `sc_fixed` и `sc_ufixed` использует статические аргументы для определения функциональности типа, в то время как типы `sc_fix` и `sc_ufix` могут использовать типы аргументов, которые будут нестатические. Статические аргументы должны быть известны во время компиляции, в то время как нестатические аргументы могут быть переменными. Типы `sc_fix` и `sc_ufix` могут использовать переменные, чтобы определить длину слова, целое слово заданной длины и т.д., а типы `sc_fixed` и `sc_ufixed` настроены во время компиляции и не меняются.

Типы `sc_fixed` и `sc_fix` указывают знаковый тип данных с фиксированной точкой. Типы `sc_ufixed` и `sc_ufix` указывают тип данных без знака с фиксированной точкой.

Объект типа с фиксированной точкой объявляется с помощью следующего синтаксиса:

- `sc_fixed<wl, iwl, q_mode, o_mode, n_bits> x;`
- `sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> y;`
- `sc_fix x(list of options);`
- `sc_ufix y(list of options);`

Аргументы для `sc_fixed` и `sc_ufixed` используются следующим образом:

`wl` - общая длина слова, используемого для представления с фиксированной точкой. Эквивалентно общему количество битов, используемых в типе.

`iwl` - целая длина слова - задает число битов, которые должны быть слева от бинарной точки (.) в числе с фиксированной точкой.

`q_mode` - режим квантования, этот параметр определяет поведение типа с фиксированной точкой, когда результат операции порождает большую точность в наименее значимых битах, чем допустимо, и как определено длиной слова и целочисленным параметром длины.

`o_mode` - режим переполнения, этот параметр определяет поведение наиболее значимых бит фиксированной точки, когда операция генерирует более высокую точность в наиболее значимых битах, чем доступно.

n_bits - количество переполненных битов, этот параметр используется только для режима переполнения и определяет, сколько бит будет переполненными, если задано поведение переполнения и происходит переполнение.

x, y - имя объекта, имя объекта с фиксированной точкой, которые объявляются.

Простой пример декларации с фиксированной точкой:

```
sc_fixed<8,4,SC_RND,SC_SAT> val;
```

Режим SC_RND будет округлять значение до ближайшего представляемого числа.

Режим переполнения SC_SAT преобразует заданное значение MAX для переполнения или MIN для нижнего состояния продукта. Максимальные и минимальные значения будут определяться из числа доступных битов. Значение MAX будет затем назначено для значения результата для положительного переполнения и MIN для отрицательного состояния переполнения

3.23. Каналы и интерфейсы

SystemC имеет встроенные механизмы, известные как каналы, для выполнения коммуникации и инкапсуляция в комплексе связей. SystemC имеет два типа каналов: примитивный и иерархический.

Примитивные каналы SystemC называются примитивными, потому что они не содержат никакой иерархии, не имеют процессов и предназначены быть очень быстрыми благодаря их простоте. Все примитивные каналы наследуются от базового класса sc_prim_channel.

SystemC содержит несколько примитивных каналов. Простейшие из них: sc_mutex, sc_semaphore, sc_fifo.

3.23.1. Канал sc_mutex

mutex является сокращением для объекта взаимного исключения. В компьютерном программировании mutex - программный объект, который позволяет нескольким программным потокам совместно использовать общий ресурс, такой как доступ к файлу, без конфликтов.

Во время разработки создается mutex с уникальным именем; впоследствии, любой процесс, который нуждается в ресурсе, должен блокировать mutex, чтобы предотвратить использования общего ресурса другими процессами. Этот процесс должен отключать mutex, когда ресурс больше не нужен. Если другой процесс пытается получить доступ к

заблокированному mutex, этот процесс преостанавливается до тех пор, пока mutex не станет доступным (разблокированным).

SystemC обеспечивает mutex через канал sc_mutex. Этот класс содержит несколько методов доступа, включая как заблокированные, так и разблокированные стили. Методы блокировки могут использоваться только в процессах SC_THREAD.

```
sc_mutex NAME;
// Блокировка мьютекса NAME (wait until)
// Разблокирована, если используется
NAME.lock();
/*Неблокирование, возвращает true, если успех,
иначе false*/
NAME.try lock()
// Чтобы освободить ранее заблокированный мьютекс
NAME.unlock();
```

Sc_mutex - это предопределенный примитивный канал, предназначенный для моделирования поведения блокировки взаимного исключения, используемого для управления доступом к ресурсу, совместно используемому параллельными процессами. mutex может быть в одном из двух эксклюзивных состояний: разблокирован или заблокирован. Только один процесс может заблокировать данный mutex за один раз. mutex может быть разблокирован только определенным процессом, который заблокировал его, но впоследствии может быть заблокирован другим процессом.

Класс sc_mutex поставляется с предопределенными методами, как показано ниже.

Int lock (): заблокируйте mutex, если он свободен, иначе подождите, пока mutex освободится.

Int unlock (): разблокировать mutex.

Int trylock (): проверить, свободен ли mutex, если свободен, то блокировать его и еще вернуть -1.

Char * kind (): Возвращает строку "sc_mutex"

Листинг 3.30

Программа sc_mutex

```
#include <systemc.h>

SC_MODULE (sc_mutex_example) {
  sc_in<bool> clock;
```

```
      sc_mutex   bus;
      int     cnt;


      void do_bus(int who) {
        cout << "@" << sc_time_stamp() <<" Bus access
by instance " << who << endl;
      }

      void do_test1() {
        while (true) {
          wait();
          cout << "@" << sc_time_stamp() <<" Checking
mutex intance 0"<<endl;
          // Проверьте, доступен ли мьютекс
          if (bus.trylock() != -1) {
            cout << "@" << sc_time_stamp() <<" Got
mutex for intance 0"<<endl;
            cnt ++;
            do_bus(0);
            wait(2);
            // Разблокировать мьютексы
            bus.unlock();
          }
          if (cnt >= 3) {
            sc_stop(); /* sc_stop запускает
завершение моделирования */
          }
        }
      }

      void do_test2() {
        while (true) {
          wait();
          cout << "@" << sc_time_stamp() <<" Checking
mutex intance 1"<<endl;
          // Подождите, пока мьютексы будут доступны
          bus.lock();
          cout << "@" << sc_time_stamp() <<" Got
mutex for intance 1"<<endl;
          do_bus(1);
```

```
        wait(3);
        // Разблокировать мьютексы
        bus.unlock();
      }
    }

    SC_CTOR(sc_mutex_example) {
      cnt = 0;
      SC_CTHREAD(do_test1,clock.pos());
      SC_CTHREAD(do_test2,clock.pos());
    }
  };

  int sc_main (int argc, char* argv[]) {
    sc_clock clock ("my_clock",1,0.5);

    sc_mutex_example  object("wait");
      object.clock (clock);

    sc_start(0); /* Первый раз вызывается
планировщик инициализации*/
    sc_start();  /* Запустите симуляцию до
появления sc_stop*/
    return 0;// Завершение симуляции
  }
```

Результаты моделирования

.

```
@1 ns Checking mutex intance 1
@1 ns Got mutex for intance 1
@1 ns Bus access by instance 1
@1 ns Checking mutex intance 0
@2 ns Checking mutex intance 0
@3 ns Checking mutex intance 0
@4 ns Checking mutex intance 0
@4 ns Got mutex for intance 0
@4 ns Bus access by instance 0
@5 ns Checking mutex intance 1
@7 ns Got mutex for intance 1
@7 ns Bus access by instance 1
@7 ns Checking mutex intance 0
@8 ns Checking mutex intance 0
@9 ns Checking mutex intance 0
@10 ns Checking mutex intance 0
@10 ns Got mutex for intance 0
@10 ns Bus access by instance 0
@11 ns Checking mutex intance 1
@13 ns Got mutex for intance 1
@13 ns Bus access by instance 1
@13 ns Checking mutex intance 0
@14 ns Checking mutex intance 0
@15 ns Checking mutex intance 0
@16 ns Checking mutex intance 0
@16 ns Got mutex for intance 0
@16 ns Bus access by instance 0
@17 ns Checking mutex intance 1
```

3.23.2. Канал sc_semaphore

Для некоторых ресурсов может быть более одной копии или владельца. Для управления этим типом ресурсов SystemC предоставляет `sc_semaphore`. При создании объекта `sc_semphore` необходимо указать, сколько копий доступны. В некотором смысле `mutex` - это просто семафор со счетом один.

Доступ `sc_semaphore` состоит из ожидания доступного ресурса и затем отправки уведомления после завершения работы с ресурсом.

```
sc_semaphore NAME(COUNT);
// Чтобы заблокировать мьютекс, NAME (wait until)
// разблокирован, если используется
NAME.wait();
/* Неблокирование, возвращает true, если успех
иначе false */
NAME.trywait()
// Возвращает количество доступных семафоров
NAME.get_value()
// Чтобы освободить ранее заблокированный мьютекс
NAME.post();
```

Важно понимать, что sc_semaphore ::wait () является явно отличающиймся методом от метода wait (), рассмотренного ранее совместно с SC_THREAD. На самом деле sc_semaphore ::wait () реализуется с помощью wait (event).

Sc_semaphore - предопределенный примитивный канал, предназначенный для моделирования поведения программного семафора, который используется для обеспечения ограниченного параллельного доступа к совместно используемому ресурсу. Семафор имеет целочисленное значение, которое устанавливается на допустимое количество одновременных обращений при создании семафора.

Sc_semaphore имеет следующие предопределенные методы.

Int wait (): если значение семафора равно 0, функция-член wait приостанавливает работу до тех пор, пока значение семафора не будет увеличено (другим процессом), после чего работа должна возобновиться и попытаться уменьшить значение семафора.

Int trywait (): если значение семафора равно 0, функция-член trywait должна немедленно вернуть значение -1 без изменения значения семафора.

Int post (): сообщение функции-члена увеличивает значение семафора. Если существуют процессы, которые приостановлены и ждут увеличения значения семафора, то только одному из этих процессов будет разрешено уменьшать значение семафора (выбор процесса является недетерминированным), в то время как остальные процессы должны снова приостанавливаться.

Int get_value (): функция-член get_value должна возвращать значение семафора.

Char * kind (): возвращает строку "sc_semaphore"

Листинг 3.31

Программа sc_semaphore

```
 #include <systemc.h>
SC_MODULE (sc_semaphore_example) {
  sc_in<bool> clock;

  sc_semaphore   bus;
  int     cnt;

  void bus_semaphore() {
    while (true) {
      wait();
```

```
          cout << "@" << sc_time_stamp() <<" Check if
semaphore is 0 " << endl;
          if (bus.get_value() == 0) {
            cout << "@" << sc_time_stamp() <<"
Posting 2 to semaphore " << endl;
            bus.post();
            bus.post();
            if (cnt >= 3) {
              sc_stop(); /* sc_stop запускает
окончание моделирования */
            }
            cnt ++;
          }
        }
      }

      void do_read() {
        while (true) {
          wait();
          cout << "@" << sc_time_stamp() <<" Checking
semaphore for intance 0"<<endl;
          // Проверьте, доступен ли семафор
          if (bus.trywait() != -1) {
            cout << "@" << sc_time_stamp() <<" Got
semaphore for intance 0"<<endl;
            wait(2);
          }
        }
      }

      void do_write() {
        while (true) {
          wait();
          cout << "@" << sc_time_stamp() <<" Checking
semaphore for intance 1"<<endl;
          // Подождите, пока не появится семафор
          bus.wait();
          cout << "@" << sc_time_stamp() <<" Got
semaphore for intance 1"<<endl;
          wait(3);
        }
      }
```

```
      SC_CTOR(sc_semaphore_example) : bus(0){
        cnt = 0;
        SC_CTHREAD(do_read,clock.pos());
        SC_CTHREAD(do_write,clock.pos());
        SC_CTHREAD(bus_semaphore,clock.pos());
      }
    };

    int sc_main (int argc, char* argv[]) {
      sc_clock clock ("my_clock",1,0.5);

      sc_semaphore_example  object("semaphore");
        object.clock (clock);

      sc_start(0); /* Первый раз вызывается
планировщик инициализации*/
      sc_start();  /* Запустить симуляцию до
появления sc_stop*/
      return 0;// Завершение симуляции
    }
```

Результаты моделирования

```
@1 ns Check if semaphore is 0
@1 ns Posting 2 to semaphore
@1 ns Checking semaphore for intance 1
@1 ns Got semaphore for intance 1
@1 ns Checking semaphore for intance 0
@1 ns Got semaphore for intance 0
@2 ns Check if semaphore is 0
@2 ns Posting 2 to semaphore
@3 ns Check if semaphore is 0
@4 ns Check if semaphore is 0
@4 ns Checking semaphore for intance 0
@4 ns Got semaphore for intance 0
@5 ns Check if semaphore is 0
@5 ns Checking semaphore for intance 1
@5 ns Got semaphore for intance 1
@6 ns Check if semaphore is 0
@6 ns Posting 2 to semaphore
@7 ns Check if semaphore is 0
@7 ns Checking semaphore for intance 0
@7 ns Got semaphore for intance 0
@8 ns Check if semaphore is 0
@9 ns Check if semaphore is 0
@9 ns Checking semaphore for intance 1
@9 ns Got semaphore for intance 1
@10 ns Check if semaphore is 0
@10 ns Posting 2 to semaphore
@10 ns Checking semaphore for intance 0
@10 ns Got semaphore for intance 0

Info: /OSCI/SystemC: Simulation stopped by user.
```

3.23.3. Канал sc_fifo

Самый популярный канал для моделирования на архитектурном уровне это `sc_fifo.` Очереди «первым пришел-первым-первым» (то есть FIFO) являются общими данными. Структура, используемая для управления потоком данных FIFO являются одной из самых простых структур.

По умолчанию `sc_fifo <>` имеет глубину 16. Тип данных элементов также необходимо указать. `Sc_fifo` может содержать любой тип данных, включая большие и сложные структуры (например, TCP/IP пакет или блок диска).

Синтаксис:

```
sc_fifo<ELEMENT_TYPENAME> NAME(SIZE);
NAME.write (VALUE);
NAME.read(REFERENCE) ;
= NAME.read () /* функциональный стиль */
if (NAME.nb_read(REFERENCE)) {
// Неблокируемая
// true, если успех
```

```
}
if (NAME.num_available() == 0)
wait(NAME.data_written_event());
if (NAME.num_free() == 0)
next_trigger(NAME.data_read_event());
```

Например, FIFO могут использоваться для буферизации данных между процессором изображений и шиной, или система связи может использовать FIFO для буферизации информационных пакетов при прохождении через сеть.

Sc_fifo - предопределенный примитивный канал, предназначенный для моделирования поведения fifo, то есть буфера first-in first-out. Fifo - это объект класса sc_fifo. Каждый fifo имеет несколько слотов для хранения значений. Количество слотов фиксируется при создании объекта. Размер слотов по умолчанию - 16.

Sc_fifo имеет следующие предопределенные методы.

Write (): этот метод записывает значения, переданные в качестве аргумента в fifo. Если fifo полный, то функция write () ожидает, пока не будет доступен слот fifo

Nb_write (): этот метод такой же, как write (); разница только в том, что когда fifo заполнен, функция nb_write () не ждет, пока не будет доступен слот fifo. Сразу возвращает false.

Read (): Этот метод возвращает самые последние записанные данные в fifo. Если fifo пуст, функция read () ожидает, пока данные не будут доступны в fifo.

Nb_read (): Этот метод аналогичен read (), только разница в том, что когда fifo пуст, функция nb_read () не ждет, пока fifo не получит некоторые данные. Сразу возвращает false.

Num_available (): Этот метод возвращает числа значений данных, доступных в fifo в текущем дельта-времени.

Num_free (): Этот метод возвращает количество свободных слотов, доступных в fifo в текущем дельта-времени.

Листинг 3.32

Программа sc_fifo

```
// EXAMPLE FIFO
// ---------

#include <systemc.h>
SC_MODULE( writer )
{
// порты
```

```
    sc_port<sc_fifo_out_if<int> > out;
    // процессы
    void main_action()
    {
    int val = 0;
    while( true ) {
    wait( 10, SC_NS ); // Ждать 10 нс
    for( int i = 0; i < 20; i ++ )
    out->write( val ++ ); // блокировка записи
    }
    }
    SC_CTOR( writer )
    {
    SC_THREAD( main_action );
    }
    };
    SC_MODULE( reader )
    {
    // port(s)
    sc_port<sc_fifo_in_if<int> > in;
    // процессы
    void main_action()
    {
    int val;
    while( true ) {
    wait( 10, SC_NS ); // ждать 10 нс
    for( int i = 0; i < 15; i ++ ) {
    in->read( val ); // блокировка чтения
    cout << val << endl;
    }
    cout << ”Available:” << in->num_available() <<
endl;
    }
    }
    SC_CTOR( reader )
    {
    SC_THREAD( main_action );
    }
    };
    int sc_main( int, char*[] )
    {
    // объявить канал (ы)
```

```
    sc_fifo<int> fifo( 10 );
    /* создать экземпляр блока (ов) и подключиться к
каналу (каналам)*/
    writer w( ”writer” );
    reader r( ”reader” );
    w.out( fifo );
    r.in( fifo );
    // Запустить симуляцию
    sc_start(40, SC_NS );
    return 0;
    }
```

Результаты моделирования

```
0               15              30
1               16              31
2               17              32
3               18              33
4               19              34
5               20              35
6               21              36
7               22              37
8               23              38
9               24              39
10              25              40
11              26              41
12              27              42
13              28              43
14              29              44
Available: 5 Available: 0   Available: 5
```

3.23.4. Иерархические каналы

Каналы бывают двух типов: примитивные и иерархические. Основное место канала - это класс, который наследуется от интерфейса. Интерфейс делает канал удобным для использования с портами. Кроме того, каналы должны наследовать либо от `sc_prim_channel` или `sc_channel`. Это различие в последних двух базовых классах - есть одна из отличительных возможностей и функций. Другими словами, `Sc_prim_channel` имеет возможности, отсутствующие в `sc_channel` и наоборот.

Примитивные каналы предназначены для обеспечения очень простой и быстрой связи. Они не содержат иерархии, портов и SC_METHODs или SC_THREAD. Примитивные каналы имеют возможность реализовать парадигму оценки-обновления. Напротив, иерархические каналы могут обращаться к портам, они могут иметь процессы и содержать иерархию, как следует из названия. Фактически, иерархические каналы

действительно являются просто модулями, реализующими один или несколько интерфейсов. Иерархические каналы предназначены для моделирования сложных коммуникационных шин таких как PCI, HyperTransport ™ или AMBA ™. Такие каналы требуют дополнительного изучения.

3.24. Коммуникации в SystemC

Рассмотрим способы подключения портов, каналов, модулей и процессов. Следующая диаграмма (рис. 3.18) иллюстрирует типы соединения, которые возможны с SystemC.

Рассмотрим фрагменты по имени и затем обсудим правила взаимосвязи.

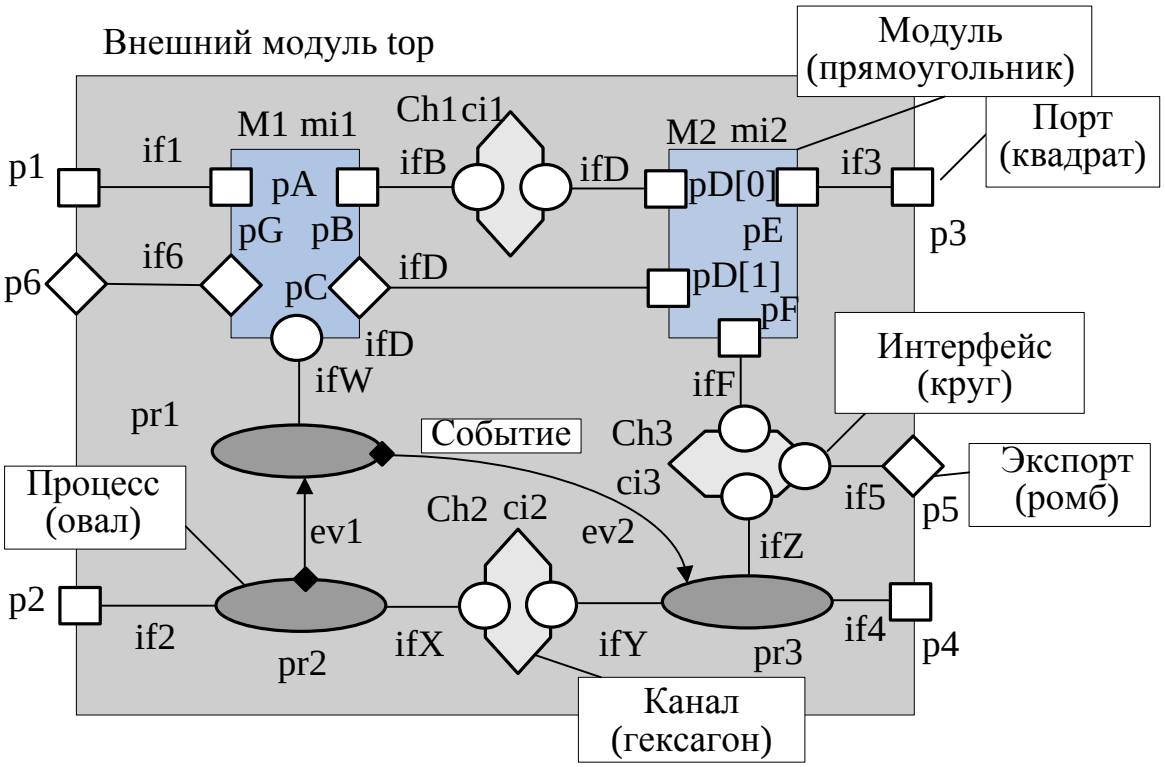


Рис. 3.18. Возможности подключения

Имеется три модуля, изображенные прямоугольниками. Внешний охватывающий экземпляр модуля имеет имя `top`. Два экземпляра подмодуля внутри `top` называются `M1 mi1` и `M2 mi2`.

Каждый из модулей имеет один или несколько портов, представленных квадратами. Направленные стрелки в портах указывают основной поток информации. Портами модуля top являются `p1, p2, p3, p4, p5` и `p6`, которые используют интерфейсы с именем `if1, if2, if3, if4, if5` и `if6` соответственно.

Портами для mi1 являются pA, pB, pC и pG, которые подключены к интерфейсам, названными if1, ifB, ifD и if6 соответственно.

Модуль M1 также предоставляет интерфейсы ifW и if6.

Порты для mi2 представляют собой pD [0], pD [1], pE и pF, которые подключены к интерфейсам с именем if3, ifD и ifF соответственно. Имеется три экземпляра каналов, представленных гексагональными формами, существуют внутри модуля top. Они называются c1i, c2i и c3i. Каждый канал реализует один или несколько интерфейсов, представленных кружками и стрелками. Стрелка предназначена для указания возможности вызова, возвращающего значение. Канал может реализовывать только один интерфейс. Канал c1i реализует интерфейсы ifB и ifD. Канал c2i реализует интерфейсы ifX и ifY. Наконец, канал c3i реализует интерфейсы if5, ifF и ifZ.

Наконец, существует три процесса с именами pr1, pr2 и pr3. Есть два явных события, ev1 и ev2, используемые для сигнализации между процессами. На этой диаграмме рис. 3.18 можно наблюдать несколько правил. Как мы уже знаем, процессы могут связываться с процессами:

• На том же уровне либо через каналы, либо через синхронизацию, либо через события

• Вне локального модуля проектирования через порты, привязанные к каналам посредством интерфейсов.

• В экземплярах подмодулей через интерфейсы к каналам, подключенным к подмодулю портов или посредством интерфейсов через сам модуль sc_export. Любая другая попытка межпроцессного общения либо запрещена, либо опасна и приводит к ошибкам.

Порты могут подключаться через интерфейсы только к локальным каналам, портам подмодулей или косвенно к портам процессов. Есть несколько интересных особенностей. Во-первых, модуль экземпляр mi1 реализует интерфейс ifW. Во-вторых, порт pD представляется массивом размера 2. Это известно как массив портов. Наконец, порт p5 и порт pC иллюстрируют sc_export.

В качестве краткого обзора давайте рассмотрим эту информацию в табличном формате (таблица 3.1).

Таблица 3.1

Способы коммуникаций

Откуда	**Куда**	**Метод**
Порт	Подмодуль	Прямое соединение через sc_port

Процесс	Порт	Прямое соединение процессом
Подмодуль	Подмодуль	Подключение локальным каналом
Процесс	Подмодуль	Подключение локальным каналом или через sc_export, или через интерфейс, выполняемый подмодулем
Процесс	Процесс	События или локальный канал
Порт	Локальный канал	Прямое соединение через sc_exportr

3.25. Моделирование уровня транзакций

Моделирование уровня транзакций - это высокоуровневый подход к моделированию цифровых систем, где детали взаимодействия между модулями отделены от деталей реализации функциональных блоков или архитектуры связи. Механизмы связи, такие как шины или FIFO, моделируются как каналы и представляются модулями с использованием классов интерфейса SystemC. Запросы транзакций выполняются посредством вызова функций интерфейса этих моделей каналов, которые инкапсулируют детали низкого уровня обмена информацией.

Моделирование на уровне транзакций также обеспечивает более высокую скорость моделирования, чем интерфейсы на основе контактов. Модели уровня транзакций могут использоваться в любое время, когда их повышенный уровень абстракции выгоден.

Моделирование уровня транзакций (TLM) выдвигается как перспективное решение выше уровня передачи регистров (RTL) в проектном потоке SoC.

Моделирование уровня трансзакций мотивировано также по следующим причинам:

- Предоставление ранней платформы для разработки программного обеспечения.
- Развертывание и проверка дизайна системного уровня.
- Необходимость использования моделей системного уровня для проверки уровня блоков.

3.26. Моделирование и отладка с помощью SystemC

После того, как вы сделали описание системы в SystemC, как правило, вы хотите моделировать его в качестве следующего шага в

процессе разработки. Здесь описывается возможности управления моделированием, предоставляемые SystemC для запуска и остановки моделирования, запроса текущего время, и понимания порядка, в котором выполняются различные процессы.

Описание системы в SystemC дает преимущество использования стандартных средства разработки C++ для компиляции и отладки. Далее описывается дополнительные средства, которые могут помочь вам в отладке SystemC программ.

3.26.1. Планировщик SystemC

Моделирования в SystemC основно на циклах: выполняемые процессы и сигналы обновляются на тактовых переходах

Библиотека SystemC включает в себя планировщик на основе цикла, который обрабатывает все события на сигналах и планирует процессы, когда соответствующие события происходят на их входах. Моделирование в SystemC следует парадигме оценки-обновления, где все процессы, которые готовы к исполнению выполняются, и только тогда их выходные сигналы обновляются.

Планировщик в SystemC выполняет следующие шаги в процессе моделирования.

1. Всем сигналам синхронизации, которые меняют свое значение в текущий момент времени, назначаются их новые значения.

2. Все SC_METHOD / SC_THREAD процессы с входами, которые изменились, выполняются. Все тело функции процесса SC_METHOD выполняется, в то время как SC_THREAD процессы не выполняются пока следующий оператор `wait()` приостанавливает выполнение процесса. SC_METHOD процессы / SC_THREAD не являются выполняемыми в определенном порядке.

3. Все SC_CTHREAD процессы, которые запускаются, имеют свои обновленные выходы, и они сохраняются в очереди, которая будет выполнена позже на шаге 5. Все выходы SC_METHOD / SC_THREAD процессов, которые были выполнены на шаге 1 также обновляются.

4. Шаги 2 и 3 повторяются до тех пор, пока сигнал не меняет свое значение.

5. Все SC_CTHREAD процессы, которые были запущены и поставлены в очередь на шаге 3, выполняются. Там нет фиксированного порядка выполнения этих процессов. Их выходы обновляются при следующем активном фронте сигнала (когда шаг 3 выполняется), и, следовательно, сохраняются внутри.

6. Время моделирования продвигается к следующему фронту тактового импульса и планировщик возвращается к шагу 1.

Если процессы взаимодействуют с помощью сигналов, порядок выполнения процесса не должен влиять на результаты моделирования. Тем не менее, если используются глобальные переменные и указатели, порядок выполнения процесса влияет на результаты моделирования. Обратите внимание, что эта семантика моделирования подобна Verilog семантике моделирования с отложенными заданиями сигналов и VHDL семантике моделирования.

3.26.2. Контроль моделирования

Вы можете начать моделирование только после того, как Вы реализуете и правильно соедините все модули и сигналы. В SystemC моделирование начинается с вызова `sc_start ()` с верхнего уровня, а именно из подпрограммы `sc_main ().` Функция `sc_start ()` принимает переменную двойного типа в качестве аргумента и моделирует систему столько стандартных единиц времени, каково значение переменной. Если вы хотите продолжать моделирование до бесконечности, для этого обеспечивают отрицательное значение для аргумента этой функции. Эта процедура генерирует все тактовые сигналы в соответствующие моменты времени и вызывает планировщик SystemC.

Моделирование можно остановить в любое время (из любого процесса) путем вызова `sc_stop ().` Функция не принимает аргументов.

Вы можете определить текущее время в процессе моделирования с помощью вызова `sc_simulation_time ()`. Эта функция возвращает текущее время моделирования в переменной двойного типа.

Чтобы помочь в отладке в процессе моделирования, переменные, порты и значения сигнала можно читать и распечатать. Печатное значение порта или сигнала является текущим значением порта или сигнала, а не просто значением, записанным в него.

3.26.3. Расширенные методы техники контроля моделирования

Есть возможность использовать другой метод для тактирования и моделирования управления, чем использование `sc_start ().` Чтобы сделать это, вы должны сначала вызвать `sc_initialize()` для инициализации SystemC планировщика. После этого вы можете установить значение сигналов, путем записи в них, и вызовом процедуры `sc_cycle (),` чтобы имитировать результат установки сигналов. Эта функция принимает переменную двойного типа в качестве аргумента. Она вызывает планировщик SystemC, моделирует, пока текущие эффекты записи сигнала не распространяются по всей системе. Затем она продвигает время моделирования на величину, заданную как аргумент

функции. Например, если время по умолчанию в модуле будет 1 нс, `sc_cycle (10)` продвигает время моделирования на 10 нс.

Для примера, предположим, что вы определили такт, как:

```
sc_clock clk(“my clock”, 20, 0.5);
```

Вы можете смоделировать генерацию тактов для единиц времени 200, вызвав по умолчанию:

```
sc_start(200);
```

В последних версиях SystemC потребуется указать размерность времени.

С другой стороны, вы можете создать тактирование самостоятельно, выполнив следующие действия:

```
sc_signal<bool> clock;
sc_initialize();
for (int i = 0; i <= 200; i++)
clock = 1;
sc_cycle(10);
clock = 0;
sc_cycle(10);
}
```

Используя эту возможность, можно вводить события асинхронно по отношению к такту в систему, как показано на следующем рис. 3.22.

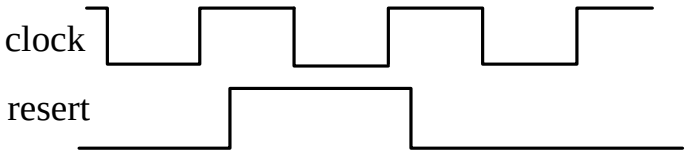


Рис. 3.22.

Для реализации этого, вы можете написать в `sc_main ():`

```
sc_initialize();
// Пусть такты работают в течение 10 циклов
for (int i = 0; i <= 200; i++)
clock = 1;
sc_cycle(10);
clock = 0;
sc_cycle(10);
}
// Ввод асинхронного сброса
clock = 1;
sc_cycle(5);
reset = 1;
sc_cycle(5);
```

```
clock = 0;
sc_cycle(10);
clock = 1;
sc_cycle(5);
reset = 0;
sc_cycle(5);
clock = 0;
sc_cycle(10);
// Теперь пусть такты работают бесконечно
for (;;)
clock = 1;
sc_cycle(10);
clock = 0;
sc_cycle(10);
}
```

Обратите внимание, что `sc_cycle ()` может быть вызван только из верхнего уровня, подобного `sc_start ().`

3.27. Трассировка осциллограмм

SystemC предоставляет функции, позволяющие создать файлы VCD (Value Change Dump), ASCII WIF (Waveform Intermediate Format), или ISDB (Integrated Data Signal Base), которые содержат значения переменных и сигналов и показывают, как они изменяются во время моделирования. Формы волны, определенные в этих файлах можно просмотреть, используя стандартные средства просмотра осциллограмм, которые поддерживают форматы VCD, WIF или ISDB.

При генерации форм волны, обратите внимание на следующее:

• Можно наблюдать только переменные, которые находятся в области видимости во время всего моделирования. Это означает, что все сигналы и элементы данных модулей можно проследить. Локальные переменные функции не могут быть прослежены.

• Переменные и скалярные сигналы, массивы и агрегатные типы можно наблюдать.

• Различные типы файлов трассировки могут быть созданы в течение того же выполнения моделирования.

• Сигнал или переменная могут быть прослежены любое количество раз в различных форматах трассировки.

3.27.1. Создание файла трассировки

Первый шаг в отслеживании сигналов создает файл трассировки. Файл трассировки, как правило, создают на высшем уровне после того, как все модули и сигналы были смоделированы. Для отслеживание формы

сигналов, используя формат VCD, файл трассировки создается с помощью вызова `sc_create_vcd_trace_file ()` с именем файла в качестве аргумента. Эта функция возвращает указатель на структуру данных, которая используется во время трассировки. Например,

```
sc_trace_file * my_trace_file;
my_trace_file =
sc_create_vcd_trace_file(“my_trace”);
```

создает файл с именем VCD `my_trace.vcd` (`.vcd` расширение автоматически добавлено). Указатель на структуру данных файла трассировки возвращается. Вы должны хранить этот указатель, чтобы он мог быть использован в вызовах подпрограмм трассировки.

Чтобы создать файл WIF, должна быть вызвана функция `sc_create_wif_trace_file ().` Для примера:

```
sc_trace_file *trace_file;
my_trace_file = sc_create_wif_file(“my_trace”);
```

создает файл с именем WIF `my_trace.awif` (`.awif` расширение автоматически добавлено). Аналогичным образом, может быть создан файл трассировки ISDB.

В конце моделирования файлы трассировки должны быть закрыты, иначе могут возникнуть ошибки. Закрыть файлы трассировки можно одной из следующих функций.

```
sc_close_isdb_trace_file(my_trace_file);
sc_close_wif_trace_file(my_trace_file);
sc_close_vcd_trace_file(my_trace_file);
```

Вызывать такую функции надо в соответствии с типом файла, который был создан. Вызывайте эту функцию как раз перед оператором возврата в вашей обычной `sc_main`.

3.27.2. Трассировка скалярной переменной и сигналов

SystemC обеспечивает трассировку функции для скалярных переменных и сигналов. Все трассировки функции имеют следующие общие черты:

• Функция называется `sc_trace ().`

• Их первый аргумент является указателем на файл трассировки структуры данных `sc_trace_file`.

• Их второй аргумент является ссылкой или указателем на переменную трассировки.

• Их третий аргумент является ссылкой на строку.

Например, далее показано, как трассируются сигнал типа `int` и переменная плавающего типа .

```
sc_signal<int> a;
float b;
sc_trace(trace_file, a, “MyA”);
sc_trace(trace_file, b, “B”);
```

В этом примере, `trace_file` является указателем типа `sc_trace_file`, который был создан ранее. `"MyA"`-это имя переменной `int`, как она будет отображаться в окне просмотра формы сигнала, и "B" - это имя переменной с плавающей точкой.

В регистрах функций трассировки создается список сигналов и переменных, которые будут прослежены. Фактическая трассировка происходит во время моделирования и обрабатывается планировщиком SystemC. Обратите внимание, что вызовы `sc_trace ()` функции выполняются только после того, как процессы и сигналы инстанцируются и после того, как файл трассировки открыт.

3.27.3. Трассировка переменных и сигналов совокупного типа

Функции трассировки, определенные в SystemC, могут принимать только сигналы или переменные скалярного типы. Для того, чтобы отслеживать переменные совокупного типа, нужно определить специальные функции трассировки для переменных этих типов, используя основные функции трассировки, которые обеспечиваются в SystemC.

Например, рассмотрим структуру

```
struct bus {
unsigned address;
bool read_write;
unsigned data;
};
```

Вам нужно определить функцию трассировки для этой структуры следующим образом:

```
void sc_trace(sc_trace_file *tf, const bus& v,
const
sc_string& NAME)
{
sc_trace(tf, v.address, NAME + “.address”);
sc_trace(tf, v.read_write, NAME + “.rw”);
sc_trace(tf, v.data, NAME + “.data”);
}
```

При вызове эта функция трассировки отслеживает структуру данных путем отслеживания отдельных полей структуры. Следует отметить, что каждому отдельному полю структуры присваивается уникальное имя, добавив имя поля к имени структуры.

3.27.4. Трассировка переменных и массивов сигналов

Для того, чтобы проследить переменную или массив сигнала, вам нужно определить специальную функцию трассировки, используя основные функции данных или трассировки сигнала, которые обеспечивает SystemC. Так, например, функцией трассировки для массивов типа `sc_signal <int>` являются

```
    void sc_trace(sc_trace_file *tf, sc_signal<int>
*v,const sc_string& NAME, int len)
    {
    char stbuf[20];
    for (int i = 0; i< len; i++) {
    sprintf(stbuf, “[%d]”, i);
    sc_trace(tf, v[i], NAME + stbuf);
    }
    }
```

Эта функция трассировки имеет один дополнительный аргумент, которым является длина отслеживаемого массива.

SystemC имеет предопределенные векторные функции для векторных типов, определенных в SystemC.

```
    (sc_int<>, sc_uinit<>, sc_biginit<>,
sc_bigunit<>, sc_lv<>, and so forth).
```

3.27.5. Отладка SystemC

Поскольку каждый поток или тактированный процесс генерирует новый поток выполнения, отладка моделирования может быть более сложной, чем в типичной линейно исполняемой программе C++. Потоки выполнения в моделировании означают, что моделирование происходит в нелинейной моде. Может быть трудно определить код, который будет выполняется следующим.

Вы можете отлаживать только код, а не библиотеки классов SystemC. Простейший способ отладки дизайна состоит в том, чтобы поместить контрольную точку в начале процесса, который требует отладки. Когда моделирование останавливается на одной из этих точек останова, моделирование остановится, и вы можете отлаживать соответствующий процесс по мере необходимости.

Глава 4. Практическое программирование в SystemC

4.1. Введение

При освоении языков программирования весьма полезно детально изучать листинги образцов реальных отлаженных программ. Отдельные фрагменты можно будет с некоторой редакцией использовать при самостоятельном составлении новых программ. Особенно полезно проверить работоспособность образцов, выполнив их загрузку в среду разработки, компиляцию и решение.

В этой главе решения примеров мы будем проводить в среде разработки Eclipse IDE, а точнее в Eclipce CDT для C/C++с компилятором Cygwin. Перед началом работы необходимо выполнить компиляцию библиотек SystemC, провести настройки Eclipse для работы с этими библиотеками. В главе 2 приведены сведения о том, как это можно сделать.

При написании этой главы использовано большое количество материалов из различных зарубежных источников в виде книг, учебных материалов и слайдов, которые автор смог найти в Интернете. Все примеры программ опробованы и подтвердили правильность. Наиболее полезные источники перечислены в библиографии.

4.2. Два основных стиля

SystemC использует два основных шаблона для программирования проектов.

Первый - это более традиционный стиль, который в значительной степени опирается на заголовки. Второй рекомендуемый стиль добавляет больше элементов в реализацию. Создание шаблонного модуля C++ обычно исключает этот стиль из-за ограничений компилятора C++.

Можно использовать любой из этих шаблонов для Вашего програмирования.

4.3. Традиционный шаблон

Традиционный шаблон, показанный на листингах 4.1 и 4.2, размещает все определения для создания примера и конструктора в файлах заголовка `(.h)`. Только реализация процессов и вспомогательных функций откладывается до создания скомпилированного файла `(.cpp).` Напомним основные компоненты в каждом файле.

Во-первых, `# ifndef / # define / # endif` предотвращает проблемы, когда заголовочный файл включается несколько раз. Использование определения `NAME_H` вполне стандартно. Это определение сопровождается включением заголовочных файлов любого подмодуля с помощью `#include`.

Затем SC_MODULE {...}; охватывает определение класса. Не забывайте конечную точку с запятой, что является довольно распространенной ошибкой. В пределах определения класса порты обычно объявляются первым, потому что они представляют интерфейс к модулю. Затем следуют локальные каналы и подмодули.

Затем мы помещаем конструктор класса и, необязательно, деструктор. Для большинства случаев для этого достаточно макроса SC_CTOR () {...}. Тело конструктора обеспечивает инициализацию, связность подмодулей и регистрации процессов. Все это будет подробно обсуждаться в примерах.

Заголовок заканчивается объявлениями процессов, помощником функций и, возможно, другими частными данными. Обратите внимание, что C++ или SystemC не диктуют порядок этих элементов в объявлении класса.

Листинг 4.1

Традиционный стиль шаблона NAME.h

```
#ifndef NAME_H
#define NAME_H
#include "submodule.h"
SC_MODULE(NAME) {
Объявления портов.
Примеры каналов / подмодулей.

SC_CTOR(NAME)
:: Инициализация
{
Связь.
Регистрация процессов}
{Декларации процесса.
Объявления помощника};
#endif
```

Тело файла реализации традиционного стиля просто включает в себя заголовочный файл SystemC, и соответствующий модуль заголовка, только что описанный. Остальная часть этого файла просто содержит внешние реализации членов функций и процессов, которые будут также описаны в примерах. Обратите внимание, что возможно отсутствие файла реализации, если в модуле нет процессов или помощника функций.

Листинг 4.2

Традиционный стиль шаблона NAME.cpp

```
#include <systemc.h>
#include "NAME.h"
NAME::Process {implementations }
NAME::Helper {implementations }
```

4.4. Рекомендуемая альтернативная форма шаблона

По разным причинам рекомендуют другой подход, который в большей степени способствует независимому развитию модулей. Эти шаблоны представлены на листингах 4.3 и 4.4 и обратите внимание на различия.

Во-первых, заголовок содержит те же `#define` и `SC_MODULE` компоненты как традиционный стиль. Различия заключаются в том, как определяются каналы / подмодули и в перемещении конструктора в тело реализации. Обратите внимание, что канал/подмодули реализуются по-разному (с использованием указателей).

Листинг 4.3

Рекомендуемый стиль шаблона NAME.h

```
#ifndef NAME_H
#define NAME_H

//Submodule forward class declarations
SC_MODULE(NAME) {
Объявления портов.
Определения Канал / Подмодуль
// Объявление конструктора:
SC_CTOR(NAME) ;
Декларации процесса.
Объявления помощника.
};
#endif
```

Листинг 4.4

Рекомендуемый стиль шаблона NAME.cpp

```
#include <systemc.h>
#include "NAME.h"
NAME::NAME(sc_module_name nm)
:: sc_module(nm)
, Инициализации
{
```

```
Распределение каналов.
Распределение субмодулей.
Связь.
Регистрация процессов.}
NAME::Process {implementations }
NAME::Helper {implementations }
```

4.5. Описание библиотек SystemC

Библиотека классов SystemC была разработана для поддержки проектирования на системном уровне. Она работает как на компьютерах, так и на платформах UNIX, и свободно загружается из Интернета.

Библиотека классов выпускается поэтапно. Первый этап - выпуск 1.0 (в настоящее время в версии 1.0.2) предоставляет все необходимые средства моделирования для описания систем, подобных тем, которые могут быть описаны с использованием языка описания аппаратных средств, такого как VHDL. В версии 1.0 представлено ядро моделирования, типы данных, подходящие для арифметики с фиксированной точкой, каналы связи, которые ведут себя как части провода (сигналы), и модули для разбивки конструкции на более мелкие части.

В версии 2.0 (в настоящее время мы используем версию 2.3.1) библиотека классов была сильно переписана, чтобы обеспечить путь обновления до истинного дизайна системного уровня. Функции, встроенные в версию 1.0, такие как сигналы, теперь построены на базовой структуре каналов, интерфейсов и портов. События были представлены в качестве примитивного средства запуска поведения вместе с набором примитивных каналов, таких как FIFO и MUTEX. Версия 2.0 позволяет добиться гораздо более мощного моделирования путем моделирования на уровне транзакций.

В будущей версии 3.0 библиотеки классов будут расширены, чтобы охватить моделирование операционных систем, чтобы поддержать разработку моделей встроенного программного обеспечения. Также возможно предоставить дополнительные библиотеки для поддержки определенной методологии проектирования. Примерами этого являются библиотека связи «Ведущий-ведомый» и SystemC Verification Library (SCV).

Библиотеки SystemC были разработана группой компаний, входящих в Open SystemC Initiative (OSCI).

Можно описать простой флип-флоп (рис. 4.1) , листинг 4.5 (www.asic-world.com) и описать сложный дизайн. SystemC - один из доступных в отрасли языков моделирования. SystemC позволяет нам

проектировать цифровые устройства на очень высоком уровне. SystemC позволяет разработчикам аппаратных средств выражать свои проекты с помощью поведенческих конструкций, отражая детали реализации на более поздней стадии проектирования в конечном проекте. С добавлением SystemC Verification, тот же язык может использоваться для проектирования, а также и для проверки.

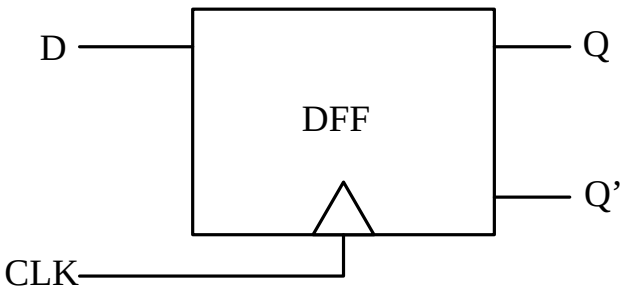


Рис. 4.1

Листинг 4.5

```
// D-FF Code
#include "systemc.h"

SC_MODULE(d_ff) {
  sc_in<bool> din;
  sc_in<bool> clock;
  sc_out<bool> dout;

  void doit() {
    dout = din;
  };

  SC_CTOR(d_ff) {
    SC_METHOD(doit);
    sensitive_pos << clock;
  }
};
```

4.6. Еще одно приветствие в SystemC

Любая книга по языку программирования обычно начинается с «Привет мир». Рассмотрим такую программу «hello world» в SystemC (листинг 4.6), решение (рис. 4.2).

Листинг 4.6

```
    /* Все модули systemc должны включать файл
заголовка systemc.h*/
    #include "systemc.h"
    // Hello_world - это имя модуля
    SC_MODULE (hello_world) {
      SC_CTOR (hello_world) {
        // Ничего в конструкторе
      }
      void say_hello() {
        // Печать «Hello World» на консоль.
        cout << "Hello World.\n";
      }
    };

    /* sc_main в функции верхнего уровня, например, в
C++ main*/
    int sc_main(int argc, char* argv[]) {
      hello_world hello("HELLO");
      // Печать “Hello world”
      hello.say_hello();
      return(0);
    }
```

Hello world.cpp

```
1  // All systemc modules should include systemc.h header file
2  #include "systemc.h"
3  // Hello_world is module name
4  SC_MODULE (hello_world) {
5    SC_CTOR (hello_world) {
6      // Nothing in constructor
7    }
8    void say_hello() {
9      //Print "Hello World" to the console.
10     cout << "Hello World.\n";
11   }
12 };
13
14 // sc_main in top level function like in C++ main
15 int sc_main(int argc, char* argv[]) {
16   hello_world hello("HELLO");
17   // Print the hello world
18   hello.say_hello();
19   return(0);
20 }
```

Console

```
<terminated> (exit value: 0) Syst-test.exe [C/C++ Application] C:\SystemC\workspace
Hello World.
```

Рис. 4.2

Любая программа в SystenC должна включать заголовочный файл “systemc.h” в начале файла. Этот файл содержит все макросы и шаблоны SystemC. Любая программа в SystemC начинается с зарезервированного слова SC_MODULE <имя_модуля>, в нашем примере строка 4 содержит модуль hello_world. В программе могут быть предпроцессорные инструкции для компилятора, такие как инструкции #include, #define перед объявлением модуля. Строка 5 содержит конструктор SC_CTOR, строка 8 содержит функцию void say_hello(). Эта функция печатает текст «Hello World» в cout<< при вызове этой функции. В SystemC, если у вас есть несколько строк внутри блока, вам нужно использовать фигурные скобки {..}. Модуль заканчивается на };, в данном случае строка 12. Нельзя забывать поставить точку с запятой.

У нас есть sc_main в строке 15, которая является функцией верхнего уровня, такой как main в C++. В sc_main мы инициализируем модуль hello_world. В строке 18 мы вызываем функцию say_hello, которая выполняет печать приветствия.

4.7. Базовый пример канала связи для сложных моделей

Это базовый пример, показывающий, как использовать SystemC, можно применять в качестве шаблона для создания более сложных моделей. Этот пример мы взяли из каталога …/Systemc-2.3.1/examples/syst/pipe.

Он состоит из 3 процессов (stage1, stage2, stage3), которые образуют отдельные этапы работы канала связи. Исходные числа формирует модуль numgen. Результаты отображает модуль Display.

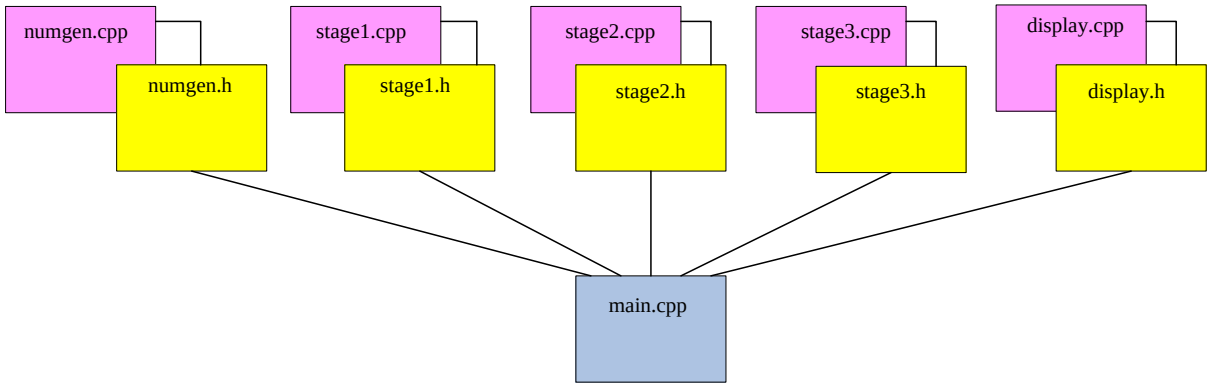


Рис. 4.3. Структура программы

В заголовочном файле numgen.h создана структура с именем numgen и определен модуль sc_module, содержащий порты out1,

out2, clock. Функция void generate () описана в конструкторе методом SC_METHOD(generate) и чувствительна к положительному фронту сигнала clock.

В исполняемом файле Numgen.cpp при каждом импульсе clock происходит вычитание чисел 1,5 и 2,8 из исходных значений 134.56 и 98.24. Результаты поступают на выходные порты out1 и out2.

Первый этап конвейера принимает сигналы с двух входов и вычисляет их сумму и разность. Второй этап принимает результаты первой стадии и вычисляет их произведение и частное. Наконец, этап 3 принимает эти выходы со второго этапа и вычисляет первый вход, возведенный в степень второго входа.

Модуль Display выполняет печать результатов.

Чтобы скомпилировать эту модель, вам нужно выполнить gmake / make. После компиляции вы должны найти исполняемый файл run.x. запустить Run.x и распечатает результаты на вашем экране.

Если Вы работаете в среде Eclipce, то все нижеприведенные файлы с листингов 4.7 – 4.17 надо скорировать в папку src, открыть файл main.cpp, выполнить компиляцию и решение. Лучше сделать это непосредственно из примера SystemC-2.3.1, скорировав файлы с расширением .h и .cpp (рис. 4.4). Из программы main.cpp надо исключить директиву #include <conio.h> и последнюю команду _getch(), которые требуются для вывода результатов в Microsoft Visual Studio.

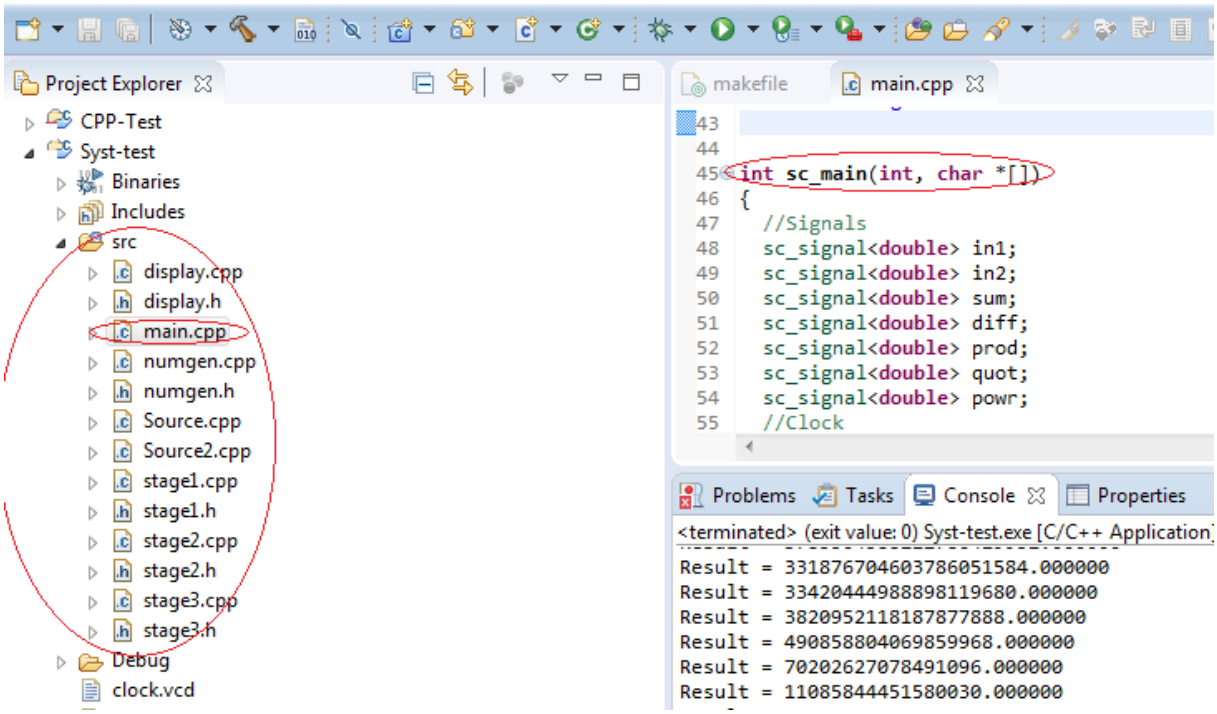


Рис. 4.4. Загрузка примера pipe в Eclipse

Программа построена по традиционному шаблону и содержит пять модулей и главную программу.

Заголовочный файл `Stage1.h` является интерфейсом для модуля `stage1`.

Листинг 4.7

Заголовочный файл `Numgen.h`

```
#ifndef NUMGEN_H
#define NUMGEN_H

struct numgen : sc_module {
    sc_out<double> out1;      //выход 1
    sc_out<double> out2;      //выход 2
    sc_in<bool>    clk;       //такт

    // метод для записи значений в выходные порты
    void generate();

    //Конструктор
    SC_CTOR( numgen ) {
    SC_METHOD( generate );   / * Объявить
генерацию как SC_METHOD и сделать его
чувствительным к положительному фронту тактового
сигнала */

        dont_initialize();
        sensitive << clk.pos();
    }
};

#endif
```

Листинг 4.8

Файл реализации Numgen.cpp

```
#include "systemc.h"
#include "numgen.h"
```

```
// Определение метода ‘generate’
void numgen::generate()
{
  static double a = 134.56;
  static double b = 98.24;

  a -= 1.5;
  b -= 2.8;
  out1.write(a);
  out2.write(b);

} // конец метода ’generate’
```

Листинг 4.9

Заголовочный файл `Stage1.h`

```
#ifndef STAGE1_H
#define STAGE1_H

struct stage1 : sc_module {
    sc_in<double> in1;   //Вход 1
    sc_in<double> in2;   //Вход 2
    sc_out<double> sum;  //Выход 1
    sc_out<double> diff; //Выход 2
    sc_in<bool>    clk;  //Такт

    void addsub();   / * метод реализации
функциональных возможностей * /
    //Конструктор
    SC_CTOR( stage1 ) {
     SC_METHOD( addsub );     / * Объявить addsub
как SC_METHOD и сделать его чувствительным к
положительному фронту тактового сигнала * /
     dont_initialize();
        sensitive << clk.pos();
    }

};
```

```
#endif
```

Листинг 4.10

Файл реализации Stage1.cpp

```
#include "systemc.h"
#include "stage1.h"

// Определение метода addsub
void stage1::addsub()
{
  double a;
  double b;

  a = in1.read();
  b = in2.read();
  sum.write(a+b);
  diff.write(a-b);

} // Конец метода addsub
```

Листинг 4.11

Заголовочный файл Stage2.h

```
#ifndef STAGE2_H
#define STAGE2_H

struct stage2 : sc_module {
    sc_in<double>  sum;      //Входной порт 1
    sc_in<double>  diff;     //Входной порт 2
    sc_out<double> prod;     //Выходной порт 1
    sc_out<double> quot;     //Выходной порт 2
    sc_in<bool>    clk;      //Такт

    void multdiv();          / * метод
обеспечения функциональности * /
    //Constructor
    SC_CTOR( stage2 ) {
```

```
    SC_METHOD( multdiv );     /* Объявить multdiv как
SC_METHOD и сделать его чувствительным к
положительному фронту тактового сигнала */
            dont_initialize();
            sensitive << clk.pos();
        }

    };

    #endif
```

Листинг 4.12

Файл реализации Stage2.cpp

```
    #include "systemc.h"
    #include "stage2.h"

    // Определение метода multdiv
    void stage2::multdiv()
    {
      double a;
      double b;

      a = sum.read();
      b = diff.read();
      if( b == 0 )
          b = 5.0;

      prod.write(a*b);
      quot.write(a/b);

    } // Конец multdiv
```

Листинг 4.13

Заголовочный файл Stage3.h

```
    #ifndef STAGE3_H
    #define STAGE3_H

    struct stage3: sc_module {
```

```
        sc_in<double>  prod;     //Входной порт 1
        sc_in<double>  quot;     //Входной порт 2
        sc_out<double> powr;     //Выходной порт 1
        sc_in<bool>    clk;      //Такт

        void power();            / *Метод  реализации
функциональных возможностей * /
        //Конструктор
        SC_CTOR( stage3 ){
            SC_METHOD( power );      / * объявить
power как SC_METHOD и сделать его чувствительным к
положительному фронту тактов * /

            dont_initialize();
            sensitive << clk.pos();      }

    };

    #endif
```

Листинг 4.14

Файл реализации `Stage3.cpp`

```
#include <math.h>
#include "systemc.h"
#include "stage3.h"

// Определение метода power
void stage3::power()
{
  double a;
  double b;
  double c;

  a = prod.read();
  b = quot.read();
  c = (a>0 && b>0)? pow(a, b) : 0.;
  powr.write(c);

} // Конец метода power
```

Листинг 4.15

Заголовочный файл DISPLAY_H

```
#ifndef DISPLAY_H
#define DISPLAY_H

struct display : sc_module {
    sc_in<double> in;         // Входной порт 1
    sc_in<bool>   clk;        // Такт

    void print_result();      / * Метод для
отображения значений входного порта * /

    //Конструктор
    SC_CTOR( display ) {
    SC_METHOD( print_result ); / * Объявить
print_result как SC_METHOD и сделать его
чувствительным к положительному фронту тактового
сигнала * /
        dont_initialize();
    sensitive << clk.pos();        }

};

#endif
```

Листинг 4.16

Файл реализации Display.cpp

```
#include "systemc.h"
#include "display.h"

#include <stdio.h>

// Определение метода print_result
void display::print_result()
{
    printf("Result = %f\n", in.read());
} // Конец метода print
```

Листинг 4.17

Главная программа Main.cpp

Это файл верхнего уровня, создающий экземпляры модулей и связывающий порты с сигналами. Эта программа формирует 50 импульсов clock с периодом 20 нс.

```
#include "systemc.h"
#include "stage1.h"
#include "stage2.h"
#include "stage3.h"
#include "display.h"
#include "numgen.h"
#include <conio.h> //для MSVC

int sc_main(int, char *[])
{
  //Сигналы
  sc_signal<double> in1;
  sc_signal<double> in2;
  sc_signal<double> sum;
  sc_signal<double> diff;
  sc_signal<double> prod;
  sc_signal<double> quot;
  sc_signal<double> powr;
  //Такты
  sc_signal<bool>   clk;

  numgen N("numgen");       / * Экземпляр модуля
‘numgen’* /
  N(in1, in2, clk );  /* Позиционное связывание
портов*/

  stage1 S1("stage1");  / * Экземпляр модуля
‘stage1’* /
  // Связывание портов по именам
  S1.in1(in1);
  S1.in2(in2);
  S1.sum(sum);
  S1.diff(diff);
  S1.clk(clk);
```

```
      stage2 S2("stage2");   /* экземпляр модуля
‘stage2’* /
      S2(sum, diff, prod, quot, clk );  /*Позиционное
связывание портов */

      stage3 S3("stage3");    /* экземпляр модуля
‘stage3’* /
      S3( prod, quot, powr, clk);   /*Позиционное
связывание портов */
      display D("display");  /* экземпляр модуля
‘display’*/
      D(powr, clk);    /*Позиционное связывание
портов */

      sc_start(0, SC_NS);   //Инициализация симуляции
      for(int i = 0; i < 50; i++){
          clk.write(1);
          sc_start( 10, SC_NS );
          clk.write(0);
          sc_start( 10, SC_NS );
      }

      _getch();  //для MSVC
      return 0;
    }
```

На рис.4.5 показан фрагмент решения в среде Eclipce .

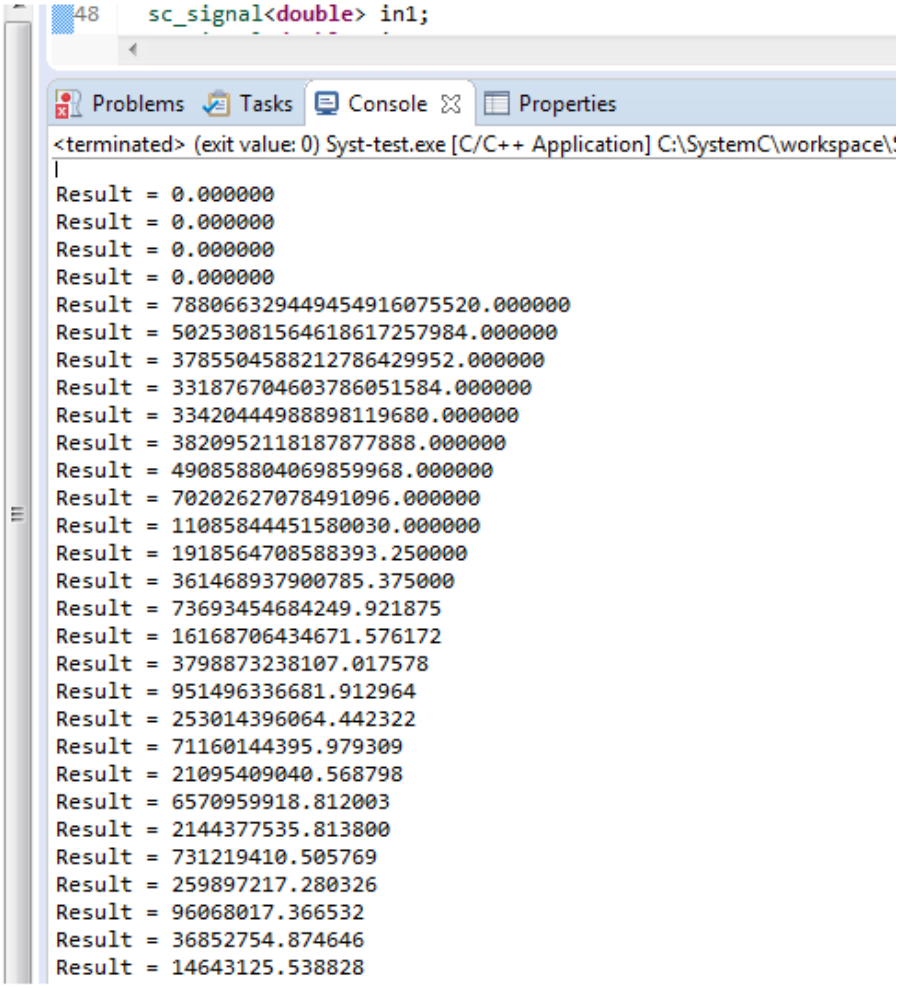


Рис. 4.5 . Фрагмент решения в среде Eclipse

На рис.4.6 показан фрагмент программы main.cpp в среде Visual Studio 2012. Для вывода результатов эта программа содержит директиву `#include <conio.h>` и оператор `_getch()`.

Решение в Visual Studio 2012 показано на рис. 4.7.

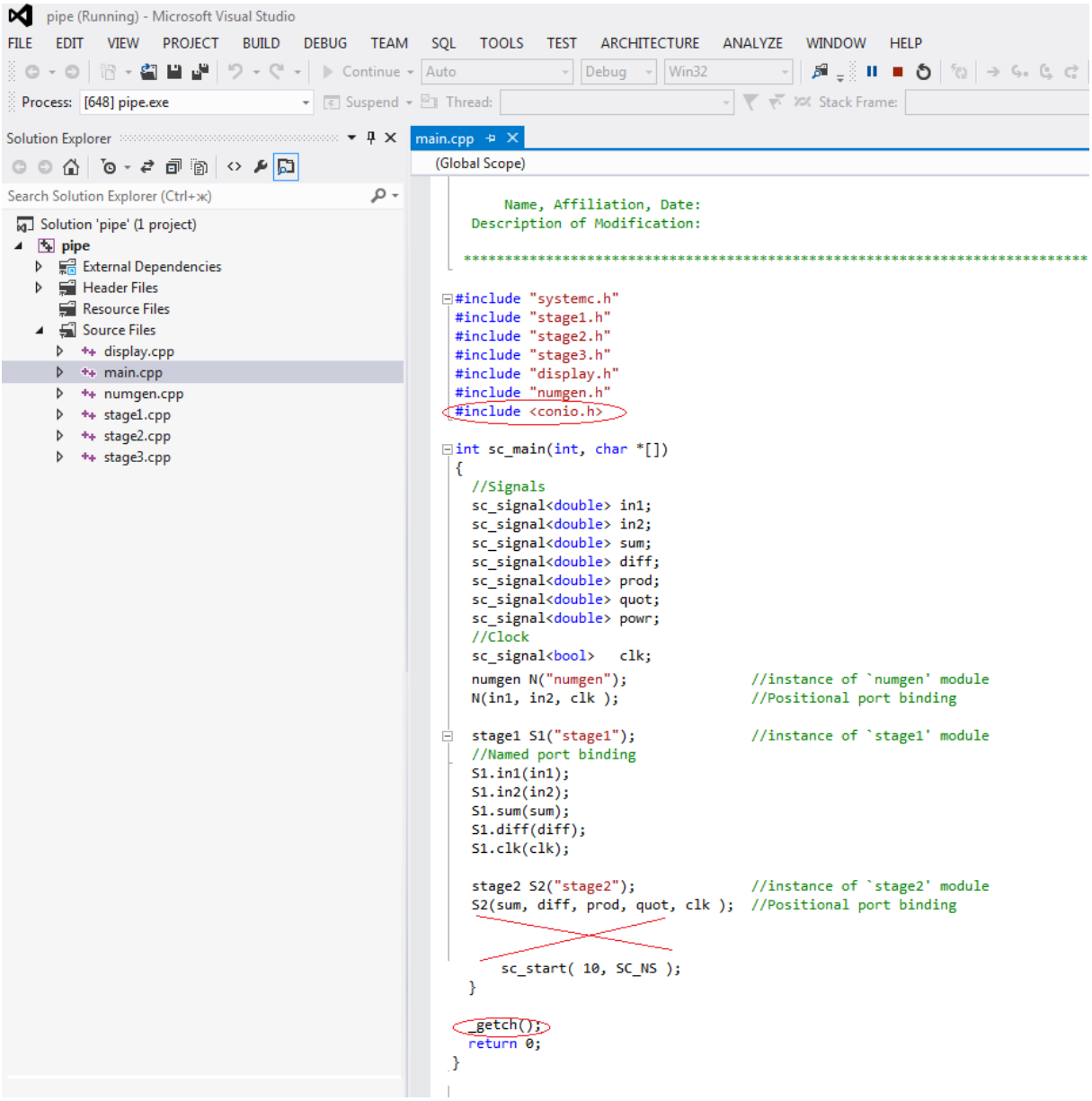


Рис. 4.6. Фрагмент программы main.cpp в среде Visual Studio 2012

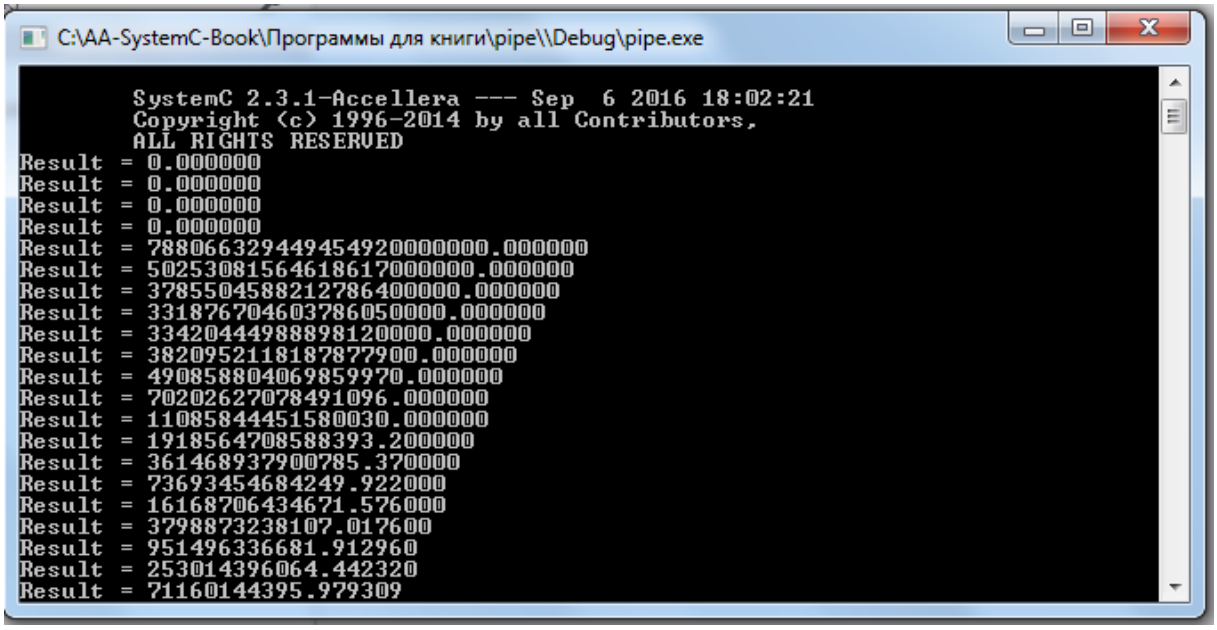


Рис. 4.7. Фрагмент решения в среде Visual Studio 2012

4.8. Использование SystemC для RTL синтеза устройств

SystemC компилятор синтезирует SystemC RTL модули или проекты с интегрированными RTL и поведенческими модулями в список соединений на уровне затворов. Он также может синтезировать системный модуль SystemC в RTL или Netlist-netlist. После синтеза вы можете использовать этот список соединений в качестве входных данных для других инструментов *компании Synopsys*, такие как Design Compiler и Physical Compiler.

Технология Synopsys лежит в основе инноваций, которые меняют то, как мы живем и работаем: интернет вещей, автономные машины, умные медицинские устройства, безопасные финансовые услуги, машинное и компьютерное зрение. Эти прорывы вступают в эпоху Smart, Secure Everything - где устройства становятся умнее, все подключено, и все должно быть безопасным.

Использование этой новой эры технологий дает усовершенствованные кремниевые чипы, которые сделаны более умными благодаря замечательному программному обеспечению, которое ими управляет. Synopsys находится на переднем крае Smart, Secure Everything с самыми передовыми в мире инструментами для проектирования кремниевых чипов, проверки, интеграции IP и тестирования безопасности приложений. Технология помогает заказчикам внедрять инновации от Silicon до Software, поэтому они могут поставлять Smart, Secure Everything.

SystemC Compiler - это инструмент, который может принимать как поведенческие, так и RTL модели SystemC и выполнять поведенческий

или RTL синтез, так как требуется, чтобы создать список соединений на уровне затворов. Вы также можете использовать SystemC компилятор для создания описания HDL для моделирования или использования с другими инструменты HDL в Вашей работе.

На рис. 4.8 показаны процессы поведенческого синтеза и RTL синтеза для описания на уровне затворов с помощью SystemC Compiler

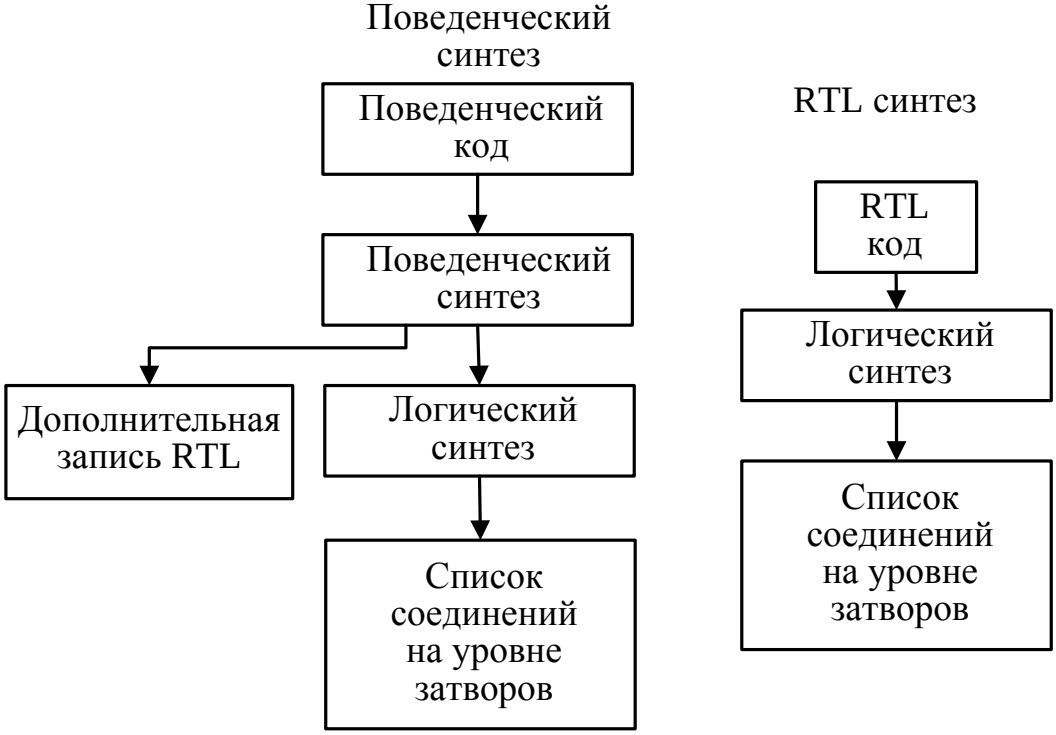


Рис. 4.8. Сравнение поведенческого синтеза и RTL-синтеза

Выбор правильной абстракции для синтеза

Вы можете реализовать аппаратный модуль, используя синтез RTL или синтез поведенческого уровня. Модель RTL описывает регистры в Вашем проекте и комбинационную логику между регистрами. Вы можете указать функциональность Вашей системы как конечного автомата (FSM-finite-state machin) и путь передачи данных. Поскольку обновления регистра привязаны к тактам, модель имеет точность цикла, как на интерфейсах, так и внутри. Точность внутреннего цикла означает, что вы указываете тактовый цикл, в котором выполняется каждая операция.

Поведенческая модель является алгоритмическим описанием. В отличие от полного программного описания поведение ввода-вывода модели описано в цикле точно. Поэтому ожидания операторов вставлено в алгоритмическое описание, чтобы четко очертить такты границ цикла и при выполнении операций ввода-вывода. В отличие от описаний RTL, поведение описывается алгоритмически, а не в терминах FSM и путей передачи данных. Оцените каждый модуль за модулем, рассмотрите каждый атрибутов модуля для определения применим ли RTL или поведенческий синтез.

Определение атрибутов, подходящих для синтеза RTL

При определении аппаратного обеспечения найдите следующие атрибуты дизайна.

Модуль, подходящий для синтеза RTL с помощью SystemC Compiler:

• Легче понять дизайн как FSM (Finite State Machine – конечный автомат) и пути передачи данных, чем как алгоритм (например, микропроцессор).

• Дизайн очень высокопроизводительный, а дизайнер, поэтому должен выполнять полный контроль над архитектурой.

• Конструкция содержит сложную память, такую как SDRAM или RAMBUS.

• Дизайн асинхронный.

Процесс синтеза RTL использует специальные команды, описанные в руководствах:

• Компилятор проектирования Synopsys

• Synopsys HDL Compiler или компилятор VHDL

• Симулятор Synopsys Scirocco VHDL

• Synopsys Verilog Compiled Simulator (VCS)

Определение атрибутов, подходящих для поведенческого синтеза

При определении аппаратного обеспечения найдите следующие атрибуты дизайна модуля, который подходит для поведенческого синтеза с SystemC

Содержание атрибутов:

• Легче понять дизайн как алгоритм, чем как FSM и путь данных (например, быстрое преобразование Фурье, фильтр, обратное квантование или цифровой процессор сигналов).

• Конструкция имеет сложный поток управления - например, есть процессор.

• У дизайна есть доступ к памяти, и вам нужно синтезировать доступ к синхронной памяти.

Обзор усовершенствований

Чистая C / C++ модель Вашего оборудования описывает только то, что составляет аппаратное обеспечение, без предоставления информации о аппаратной структуре или архитектуре. Начиная с модели C/C++, целью

первого этапа разработки является создание аппаратной структуры. Чтобы синтезировать оборудование, Вам необходимо:

• Определить порты ввода-вывода для аппаратного модуля;

• Указать внутреннюю структуру как модулей;

• Указать внутреннюю связь между модулями.

Для каждого блока в дизайне Вы начинаете с функционального уровня SystemC и уточняете его в RTL-модели для синтеза с компилятором SystemC.

Этапы усовершенствования модели высокого уровня в RTL-модель для синтеза включают следующие шаги:

• Определить ввод / вывод в точном соответствии с циклом;

• Отделить логику управления и пути данных;

• Определение архитектуры путей данных;

• Определить явные FSM для логики управления.

Высокоуровневая система SystemC может содержать абстрактные порты, типы которых не могут быть переведены на оборудование. Для каждого абстрактного порта, вам необходимо определить порт или набор портов для замены каждого терминала абстрактного порта и заменить все обращения к абстрактному порту или терминалу с доступом к вновь определенным портам.

Входы и выходы для синтеза RTL

Для компилятора SystemC требуется иметь описание SystemC RTL, технологическую библиотеку и синтетическую библиотеку.

На рис. 4.9 показан процесс синтеза в компиляторе SystemC и выход из него.

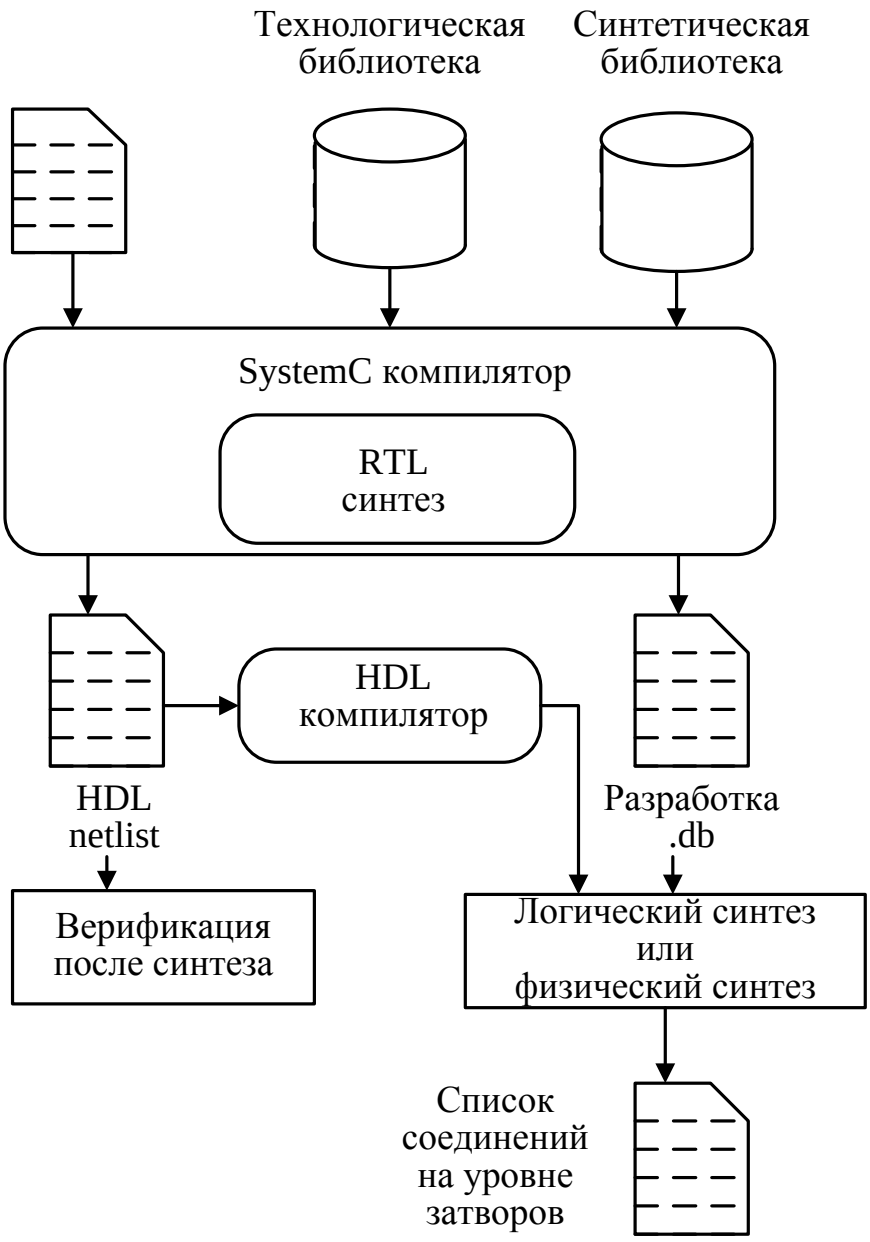


Рис. 4.9. Процесс RTL- синтеза в SystemC

Описание SystemC RTL делают, используя SystemC Class Library. Примеры конструкций будут рассмотрены далее.

Технологическая библиотека предоставляется поставщиком ASIC в Synopsys .db Format базы данных. Поставщик предоставляет области, сроки, модели загрузки соединений и условия эксплуатации. Вы предоставляете путь к выбранной вами Библиотеке технологий для Вашего дизайна, определяя целевую библиотеку.

Синтетическая библиотека является технологически независимой библиотекой логики и включает такие компоненты, как сумматоры и множители. Компилятор SystemC сопоставляет ваши проектные операторы с синтетической библиотекой логических компонентов. Вы предоставляете

путь к выбранным Вами синтетическим библиотекам для Вашего дизайна, определяя переменную синтетическую библиотеку в dc_shell.

SystemC Compiler создает расширенный .db-файл для ввода в инструмент проектирования компилятора. Он также генерирует RTL-файлы HDL (например, Verilog), которые могут использоваться в потоках на основе HDL.

4.9. Испытательные программы Testbench

Testbench (испытательный стенд) – это модель, которую используют для испытания и верификации проектов путем тестирования. В добавление к написанию проекта в SystemC можно написать testbench, используя разные языки, с том числе и SystemC.

Testbench имеет три главные цели:

1. Генерировать стимулы для моделирования (осциллограммы).

2. Подавать стимулы на тестируемое устройство и собирать выходные реакции.

3. Сравнивать выходные реакции с ожидаемыми результатами.

Есть много различных путей для написания testbench. Стимулы могут быть записаны в модуле stimulus.h и stimulus.cpp, проверка выходных результатов и сравнение можно записать в другом моделе monitor.h и monitor.cpp. Тестируемый проект будет в отдельном модуле dut.h и dut.cpp. Главная программа соединяет различные модули и подключает их к сформированному испытательному стенду. Другой путь состоит в дополнении генерации стимулов в главную программу во время проверки функционирования.

Рекомендуемы стиль создания конструкции testbench следующий. Сигналы и такты должны быть декларированы в первую очередь, они могут следовать за содержанием модуля. Затем должны быть открыты файлы трассировки и сделан вызов трассировки И, наконец, могут быть вызваны команды моделирования start и stop. Открытые файлы трассировки должны быть закрыты после окончания моделирования.

4.9.1. Основные конструкции испытательных программ

SystemC обеспечивает следующие конструкции, помогающие в моделировании:

`Sc_clock`: генерирует тактовый сигнал;

`Sc_trace`: сохраняет информацию трассировки в файл специального формата;

`Sc_start`: запускает симуляцию на определенное время;

`Sc_stop`: останавливает симуляцию;

`Sc_time_stamp`: получает текущее время моделирования с единицами времени;

`Sc_simulation_time`: получает текущее время моделирования без единиц времени.

`Sc_cycle, sc_initialize`: используется для выполнения циклического моделирования.

`Sc_time`: определяет значение времени.

Рассмотрим более детально эти конструкции, возможно, повторив пройденный материал.

`Sc_clock`

Позволяет создавать специальные тактовые объекты, которые содержат временные осциллограммы.

Объявление `clock`

```
Sc_clock rclk (“rclk”, 10, SC_NS);
```

создает тактовую осциллограмму с периодом 10 нс, с рабочим циклом 50% и начальным значением `true (1)`.

Другая декларация:

```
Sc_clock mclk (“mclk”, 10, SC_NS, 0.2, 5, SC_NS,
false);
```

Создает тактовую осциллограмму с периодом 10 нс, рабочим циклом 20%, первый форнт появляется после 5 нс и начальное значение на первом фронте `false (0)`.

Первый аргумент – это имя тактов и оно должно быть объявлено. По умолчанию период равен 1 единице времени (по умолчанию 1 нс).

Например: `sc_clock clka (“clka”);`

- это тактовый сигнал с периодом 1 нс, раблчим циклом 50%, начальным значением true и первым фроном в момент 0.

`Sc_trace`

SystemC поддерживает три различных формата, в которых можно сохранять результаты моделирования:

VCD (Value Change Dump);

WIF (Waveform Interchange Format);

ISDB (Integrated Signal Data Base).

Чтобы сохранить значения в формате VCD, используют соответствующую функцию вызова: `Sc_create_vcd_trace_file()`. Расширение `.vcd` добавляется автоматически.

Заголовок файла декларирует тип указателя для `sc_trace_file`.

Например:

```
sc_trace_file *tfile=sc_create_vcd_trace_file
(“myvcddump”);
```

Эта декларация создает VCD файл трассировки, названный `myvcddump.vcd`. Укакзатель, который надо использовать в testbench, `tfile`.

Используя функцию `sc_trace()` , вы можете специфицировать сигналы, значения которых вы хотите сохранить в файле трассировки.

Пример: `sc_trace (tfile, signal_name, “signal_name”);`

Эта функция сохраняет значение `signal_name` в файле трассировки `tfile.` Строка значений сигнала будет появлятся в файле трассировки.

Перед выходом из программы `sc_main` испытательного стенда надо закрыть файлы трассировки:

```
Sc_close_vcd_trace_file (pointer_to_trace_file);
```

Вызов функции `sc_trace()` должен появится после нового открытия файла трассировки и создания трассируемых сигналов.

Sc_start ()

Этот метод указывает ядру моделирования начать симуляцию. Он может быть специфицирован после всех конкретизаций и после вызова трассировок.

Пример:

```
Sc_start (100, SC_MS);
```

говорит ядру моделирования запустить симуляцию на 100 мс.

Метод вызова: `sc_start (-1);`

говорит симулятору запустить моделирование навсегда.

Sc_stop ()

Метод `sc_stop ()` может быть использован в любых процессах, чтобы остановить моделирование:

```
Sc_stop ();
```

- не имеет каких-либо аргументов.

Sc_time_stamp

Этот метод возвращает текущее время моделирования с единицами измерения.

Например:

```
    cout <<"Current time is" <<sc_time_stamp
()<<endl;
```

напечатает, например : Current time is 25 ns

Sc_simulation_time

Этот метод возвращает текущее время симуляции как целое значение двойной длины в терминах единиц времени по умолчанию. Наример:

```
    Double curr_time=sc_simulation_time ();
```

где выполняется, что `curr_time` будет содержать текущее время моделирования.

Sc_cycle и sc_initialize

Эти два метода используются для выполнения циклического моделирования. Наример, если Вы хотите оценивать Ваш проект через каждые 10 единиц времени, Вы должны делать циклическую симуляцию. В таком случае Вы не используете `sc_start`, но используете `sc_initialize()` и `sc_cycle ().`

Метод `sc_initialize ()` инициализирует ядро симуляции.

Метод `sc_cycle ()` выполняет все процессы , которые готовы к запуску и которые могли взять некоторое количество дельта-циклов, пока нет других процессов, готовых к запуску. Затем этот метод трассирует необходимые сигналы перед расширением времени моделирования на указанную величину. Так:

```
    Sc_cycle (10, SC_US); // 10 microsecond
```

будет моделировать все процессы и затем увеличит время моделирования на 10 мкс.

Sc_time

Декларацию `sc_time` используют, чтобы задать значение времени, например, 10 нс, 20 нс. Объект `sc_time` может быть затем использован всюду, где требуется значение времени, как в `sc_clock` и `sc_start`.

Например:

```
    Sc_time t1 (100, SC_NS);  /* Задает значение 100
нс*/
    Sc_time t2 (20, SC_PS); /*Задает значение 20 пс*/


    // Следующие две формы эквивалентны:
    Sc_start (t1); // Запустить симуляцию за 100 нс
    Sc_start (100, SC_NS);
```

```
Sc_cycle (t2);

Sc_time period (10, SC_NS);
Sc_time start_time (2,SC_NS);
Sc_clock fclk ( “fclk”, period, 0.2, start_time,
true);
```

Можно использовать одну из единиц времени: `SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC.`

По умолчанию время разрешения 1 ps. Оно может быть изменено, используя метод `sc_set_time_resolution()`, например так:

```
sc_set_time_resolution(100, SC_PS);
```

Временное разрешение будет 100 ps.

Если Вы установите:

```
Sc_clock c1 (“c1”, 20.26, SC_NS);
```

то будут созданы такты с периодом 20300 ps. Временное разрешение можно задавать только кратным 10 и обычно оно задается только один раз в самом начале программы `sc_main ()`.

4.9.2. Сигналы

Регулярные периодические сигналы (waveforms), такие как такты, можно генерировать, используя декларацию `sc_clock`. Рассмотрим различные способы создания стимулов.

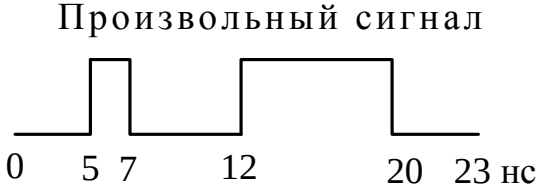


Рис. 4.10. Форма произвольного сигнала

Можно использовать процесс SC_THREAD. Этот процесс можно приостанавливать и перезапускать, основываясь на времени или возникающий событиях. Вот модуль, который генерирует произвольный сигнал:

```
//File: wave.h
#include “systemc.h”

SC_MODULE (wave) {
sc_out<bool> sig_out;
void prc_wave ();
```

```
SC_CTOR (wave) {
SC_THREAD (prc_wave);
};


//File: wave.cpp
#include “wave.h”

void wave::prc_wave() {
sig_out=0;
wait (5, SC_NS);
sig_out=1;
wait (2, SC_NS);
sig_out = 0;
wait (5, SC_NS);
sig_out=1;
wait (8, SC_NS);
sig_out = 0;
}
```

Процесс `prc_wave` начнет исполнение во время инициализации. Все процессы (SC_METHOD, SC_THREAD) выполняются один раз во время инициализации прямо перед началом симуляции. Перед началом выполнения выходному значению присвоен 0. Последующая команда ожидания `wait` приостанавливает процесс на 5 нс. Такая задержка не может появится в процессе SC_METHOD. После задержки выходное значение становится равным 1 и процесс задерживается на 2 нс и т.д.

Сложный повторяющийся сигнал

Если Вы хотите повторять сигнал каждые 100 нс, это можно сделать, используя бесконечный цикл в процессе SC_THREAD.

```
// File: wave2.cpp
#include “wave.h”

void wave::prc_wave(){
while (1) {
sig_out=0;
wait (5, SC_NS);
sig_out=1;
wait (2, SC_NS);
sig_out=0;
wait (5, SC_NS);
```

```
sig_out=1;
wait (8, SC_NS);
sig_out=0;
wait (80, SC_NS);
}
}
```

Тактовый сигнал можно также моделировать следующим образом.

```
// File: myclock.h
#include “systemc.h”
const int start_value=0;
const int INITIAL_DELAY=5;
const int FIRST_DELAY=2;
const int SECOND_DELAY=3;

SC_MODULE (myclock) {
Sc_out<bool>clk_out;
void prc_myclock();

SC_CTOR (myclock) {
SC_THREAD (prc_myclock);
}
};

// File: myclock.cpp
#include “myclock.h”

Void myclock::prc_myclock() {
Clk_out=START_VALUE;
Wait  (INITIAL_DELAY, SC_NS);

While (1) {
Clk_out=!clk_out;
Wait (FIRST_DELAY, SC_NS);
Clk_out=!clk_out;
Wait (SECOND_DELAY, SC_NS);
}
}
```

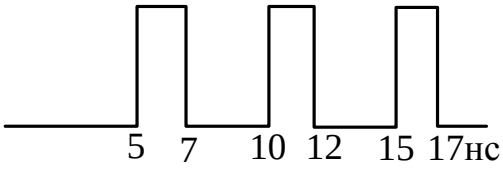


Рис. 4.11. Сигнал собственного генератора

Генерация произвольных импульсов

Рассмотрим модель генератора импульсов, синхронизированных тактовым сигналом.

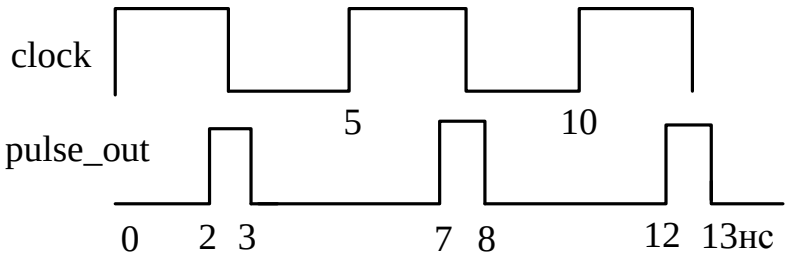


Рис. 4.12. Форма импульсного сигнала

Листинг 4.18

```
//File: pulse.h
#include “systemc.h”
#define DELAY 2, SC_NS
#define ON_DURATION 1, SC_NS

SC_MODULE (pulse) {
Sc_in<bool>clk;
Sc_out<bool>pulse_out;

Void prc_pulse();

SC_CTOR (pulse){
SC_THREAD (prc_pulse);
Sensitive_pos<<clk;
}
};

//File: pulse.cpp
#include “pulse.h”

Void pulse::prc_pulse(){
Pulse_out=0;
While (true) {
```

```
Wait (); // Подождать положительного  фронта такта
Wait (DELAY);
Pulse_out=1;
Wait (ON_DURATION);
Pulse_out=0;
}
}

// File: pulse_main.cpp
#include “pulse.h”

Int sc_main (int argc, char *argv[]) {
Sc_signal<bool>pout;
Sc_trace_file *tf;
Sc_clock clock (“masater_clk”, 5, SC_NS);

// Создание импульсного модуля:
pulse p1 (“pulse_p1”);
p1.clk (clock(;
p1.pulse_out (pout);



/* Определить трассировочный файл pulse.vcd и
сигналы трассировки:*/
tf=sc_create_vcd_trace_file (“pulse”);
sc_trace (tf, clock, “clock”);
sc_trace (tf, pout, “pulse_out”);

sc_start (100, SC_NS);
sc_close_vcd_trace_file (tf);
cout<<”Finished at time” <<sc_time_stamp()<<endl;
return 0;
}
```

Процесс SC_THREAD может опционально иметь лист чувствительности. Команда `wait` в процессе должна ждать изменения сигнала из списка чувствительности. Это первая команда в цикле. Две другие команды ожидения выполняют только истекщее время и на связаны с листом чувствительности.

Реактивные стимулы

Реактивные стимулы генерируют на основе текущего состояния проекта, который тестируется, и testbench, тоже реактивная, основывается на состоянии проекта. Такое приближение полезно, если разные стимулы надо прикладывать в разных состояниях проекта. Такой подход полезен, если требуются различные стимулы, когда проект находится в различных стадиях. На рис. 4.13 показан механизм взаимодействия между устройством и испытательной моделью. Входной сигнал `resert` сбрасывает модель расчета факториала в исходное состояние. Сигнал `start` устанавливается после появления входных данных `data`. Когда вычисления выполнены, выходной сигнал `done` показывает, что вычисленный сигнал появляется на выходах `fac_out` и `exp_out.` Результирующее значение факториала будет (fac_out*2 $^{exp\_out}$). Модель testbench предоставляет входной сигнал `data` от начального значения 1 до 20 с инкркментом 1. Testbench выдает входные данные, устанавливает сигнал `start`, ожидает сигнала `done`, и затем выдает следующее входное значение. Подтверждения используются, чтобы убедиться, что появившиеся на выходе значения корректны.

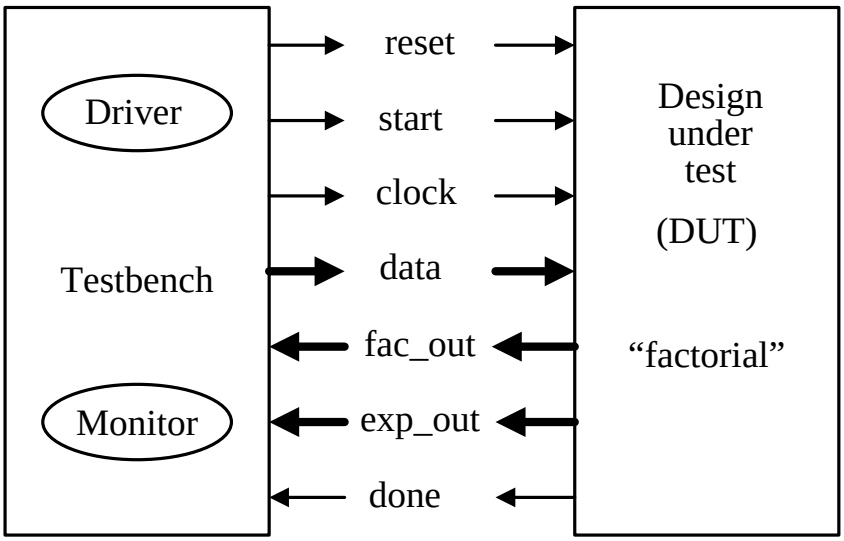


Рис. 4.13. Взаимодействие между testbench и тестируемым проектом

4.10. Пример моделирования D-триггера с испытательной программой

D-триггер (рис. 4.14) имеет вход сброса (`reset`), тактовый вход (`clock`), вход данных (`data`) и выход (`data_out`). Испытательная программа `testbench` использует метод `wave()`, чтобы создать последовательность входных сигналов. Метод `wait()` приостанавливает процесс и ждет появления события из его листа статической чувствительности (лист

описан в блоке конструктора). В этом примере этим событием является положительный фронт сигнала `clk`. Модуль `check` просто передает выходные изменения на выходной терминал.

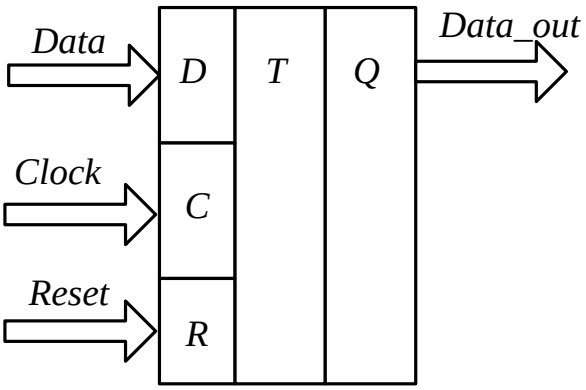


Рис. 4.14. Схема D-триггера

Листинг 4.19

Программа ff.h

```
//File: ff.h Заголовочный файл модуля триггера

#include "systemc.h"
SC_MODULE(ff){

    sc_in<bool> clk;
    sc_in<bool> reset;
    sc_in<int> data;
    sc_out<int> data_out;

    void prc_ff();
    SC_CTOR (ff)
    {
        SC_METHOD (prc_ff);
        sensitive_pos<<clk<<reset;
    }
};
```

Листинг 4.20

Программа ff.cpp

```
//File:ff.cpp Исполняемый файл модуля триггера
#include "ff.h"
```

```
void ff::prc_ff() {
    if (reset)
        data_out=0;
    else
        data_out=data;
}
```

Листинг 4.21

Программа ff_tb.h

```
//File:ff_tb.h Заголовочный файл testbench
#include "systemc.h"


SC_MODULE(ff_tb) {
    sc_in<bool>clk;
    sc_in<int> data_out;
    sc_out<bool>reset;
    sc_out<int> data;
    void test();
    void check();

    SC_CTOR(ff_tb){
        SC_THREAD(test);
        sensitive_pos<<clk;
        SC_METHOD(check);
        sensitive<<data_out;
    }
};
```

Листинг 4.22

Программа ff_tb.cpp

```
//File:ff_tb.cpp Исполняемый файл testbench
#include "ff_tb.h"

void ff_tb::test(){
    reset.write(1);
    wait(20, SC_MS);
```

```
    reset.write(0);
    data.write(1);
    wait(20, SC_MS);
    data.write(0);
    wait(20, SC_MS);
    data.write(1);
    wait(20, SC_MS);
    data.write(0);
    wait(20, SC_MS);
    data.write(1);
    wait(20, SC_MS);
    data.write(1);
    wait(20, SC_MS);

}

void ff_tb::check() {
    cout<<"Output data is (@"<<sc_time_stamp()<<
    "):"<<data_out.read()<<endl;
}
```

Листинг 4.23

Программа main.cpp

```
//File:main.cpp Файл главной программы
#include "ff.h"
#include "ff_tb.h"


int sc_main(int argc, char *argv[]){
    sc_clock clk ("clk", 2, SC_MS);
    sc_signal<bool> reset;
    sc_signal<int> data, data_out;

    ff_tb tb("tb");
    tb.clk(clk);
    tb.reset(reset);
    tb.data_out(data_out);
    tb.data(data);

    ff f1 ("ff_f1");
```

```
        f1.clk(clk);
        f1.reset(reset);
        f1.data_out(data_out);
        f1.data(data);
            /* Начать симуляцию и продолжать до
остановки моделирования из-за выполнения sc_stop () в
модуле ff_tb.*/
        sc_start(200, SC_MS);
        return 0;
    }
```

Результаты моделирования

```
Info: (I804) /IEEE_Std_1666/deprecated: sc_sensitive_pos
Output data is (@0 s):0
Output data is (@20 ms):1
Output data is (@40 ms):0
Output data is (@60 ms):1
Output data is (@80 ms):0
Output data is (@100 ms):1
```

Рис. 4.15

4.11. Программы «First_counter» и «Testbench»

На рис. 4.16 показана схема счетчика, который надо запрограммировать в SystemC и проверить функционирование. Исполняемая программа работы счетчика `first_counter.cpp` приведена на листинг 4.24.

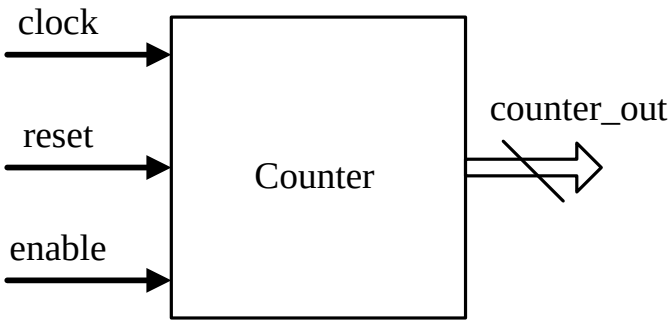


Рис. 4.16. Счетчик

Листинг 4.24

Программа first_counter.cpp

```
#include "systemc.h"

SC_MODULE (first_counter) {
```

```
      sc_in_clk     clock ;   // Входной такт проекта
      sc_in<bool>   reset ;      /* Активный высокий,
синхронный вход сброса */
      sc_in<bool>   enable;      /* Активный высокий
разрешающий сигнал для счетчика*/
      sc_out<sc_uint<4> > counter_out; /* 4-битный
векторный выход счетчика */

      //---- Локальные переменные здесь -----------
      sc_uint<4> count;

      //--------- Код начинается здесь -----------
      /* Ниже функция реализует фактическую логику
счетчика */
      void incr_count () {
        /* На каждом фронте такта мы проверяем,
активен ли сброс */
        /* Если активен, мы загружаем в счетчик
4'b0000*/
        if (reset.read() == 1) {
          count =  0;
          counter_out.write(count);
        /* Если включена функция enable, то мы
увеличиваем счетчик */
        } else if (enable.read() == 1) {
          count = count + 1;
          counter_out.write(count);
          cout<<"@" << sc_time_stamp() <<" ::
Incremented Counter "
             <<counter_out.read()<<endl;
        }
      } // Конец функции incr_count

      // Конструктор для счетчика
      /* Поскольку этот счетчик является
чувствительным к положительному фронту такта,  мы
запускаем нижний блок по положительному  фронту
такта, а также, когда сброс изменяет состояние */
      SC_CTOR(first_counter) {
        cout<<"Executing new"<<endl;
        SC_METHOD(incr_count);
        sensitive << reset;
```

```
        sensitive << clock.pos();
    } // Конец конструктора

}; // Конец модуля счетчика
```

В этой программе введен и описан модуль `SC_MODULE (first_counter)`. Модуль имеет три входных порта `(clock, resert, enable)`, выходной порт `(counter_out)`. Локальной переменной является 4-х битный вектор `counter_out`.

Функция `void incr_count ()` выполняет инкрементацию счетчика. Вначале счетчик обнуляется. Если сигнал `resert` активный, каждый нарастающий фронт тактового сигнала `clock` будет вызвать сброс счетчика. При этом, если сигнал `resert` неактивный, а сигнал `enable` активный, происходит инкрементация счетчика на единицу. Выводится время с начала моделирования и значание числа в счетчике.

Далее программа содержит конструктор `SC_CTOR(first_counter)`, в котором описан процесс `SC_METHOD (incr_count)` и чувствительность метода к сигналам `resert` и положительному фронту сигнала `clock.`

Любая цифровая схема, независимо от того, насколько сложна она, должна быть проверена. Для логики счетчика нам нужно предоставить логику синхронизации и сброса. Как только счетчик выходит из сброса, мы переключаем вход разрешения на счетчике и проверяем форму волны, чтобы проверить, правильно ли подсчитывает счетчик. То же самое делается в SystemC. Схема программы из www.asic-world.com показана на рис. 4.17.

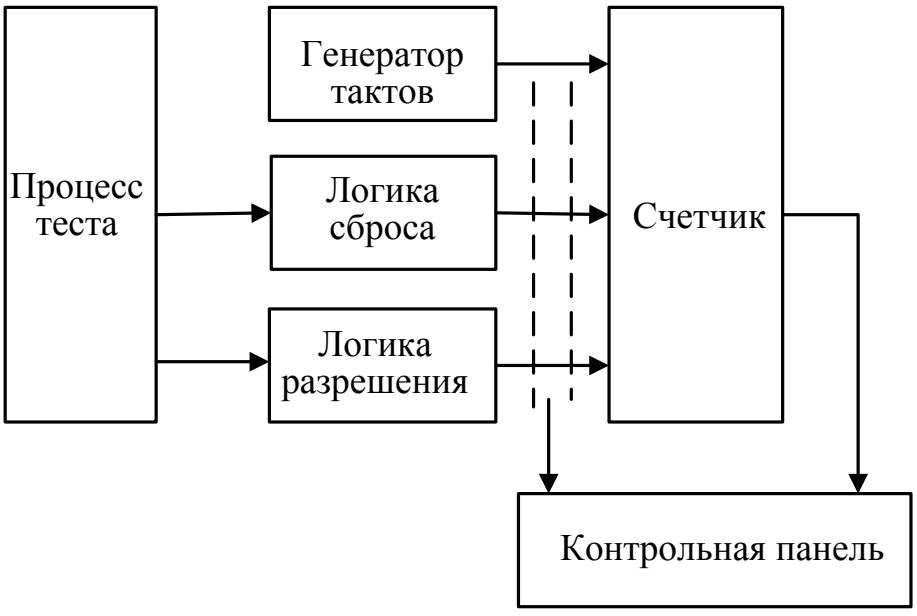


Рис. 4.17. Схема программы Testbench

Программа testbench состоит из генератора тактовых импульсов, управления сбросом, управления разрешением включением и логики монитора / контролера. Ниже на листинге 4.25 приведен простой код `testbench` без логики `monitor / checker`. Рекомендуем, пользуясь комментариями, внимательно изучить программу, скорировать файлы `first_counter.cpp` и `first_counter_tb.cpp` в папку `src` проекта в среде Eclipse C++, выполнить компиляцию и решение.

Листинг 4.25

Программа first_counter_tb.cpp

```
#include "systemc.h"
#include "first_counter.cpp"

int sc_main (int argc, char* argv[]) {
  sc_signal<bool>   clock;
  sc_signal<bool>   reset;
  sc_signal<bool>   enable;
  sc_signal<sc_uint<4> > counter_out;
  int i = 0;
  // Подключить DUT
  first_counter counter("COUNTER");
    counter.clock(clock);
    counter.reset(reset);
    counter.enable(enable);
    counter.counter_out(counter_out);

  sc_start(1, SC_MS);

  // Открыть VCD файл
  sc_trace_file *wf =
sc_create_vcd_trace_file("counter");
  // Дамп требуемых сигналов
  sc_trace(wf, clock, "clock");
  sc_trace(wf, reset, "reset");
  sc_trace(wf, enable, "enable");
  sc_trace(wf, counter_out, "count");

  // Инициализировать все переменные
  reset = 0;       // initial value of reset
  enable = 0;      // initial value of enable
```

```
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 1;    // Установить сброс
      cout << "@" << sc_time_stamp() <<" Asserting
reset\n" << endl;
      for (i=0;i<10;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      reset = 0;    // Отменить сброс
      cout << "@" << sc_time_stamp() <<" De-Asserting
reset\n" << endl;
      for (i=0;i<5;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" Asserting
Enable\n" << endl;
      enable = 1;  // Установить разрешение
      for (i=0;i<20;i++) {
        clock = 0;
        sc_start(1, SC_MS);
        clock = 1;
        sc_start(1, SC_MS);
      }
      cout << "@" << sc_time_stamp() <<" De-Asserting
Enable\n" << endl;
      enable = 0; // Отменить разрешение

      cout << "@" << sc_time_stamp() <<" Terminating
simulation\n" << endl;
      sc_close_vcd_trace_file(wf);
      return 0;// Завершить симуляцию
```

}

Результаты решения показаны на рис. 4.18.

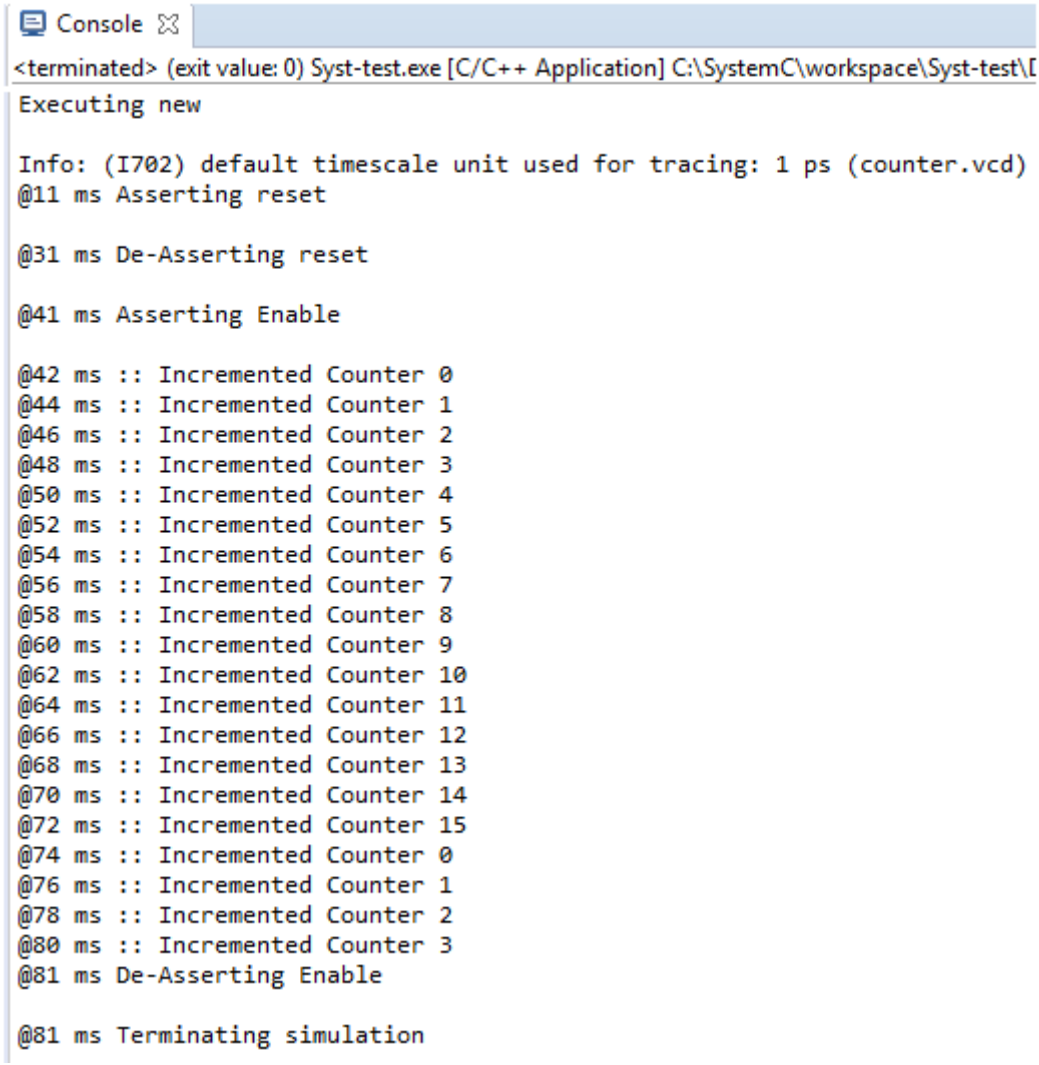


Рис. 4.18. Результаты ммоделирования первого счетчика

Программа `first_counter_tb.cpp` создает VCD – файл и выполняет трассировку сигналов `clock, resert, enable, count` (листинг 4.26).

Листинг 4.26

```
  // Открыть VCD файл
  sc_trace_file *wf =
sc_create_vcd_trace_file("counter");
  // Дамп требуемых сигналов
  sc_trace(wf, clock, "clock");
  sc_trace(wf, reset, "reset");
  sc_trace(wf, enable, "enable");
  sc_trace(wf, counter_out, "count");
```

По умолчанию файл VCD записан в папке Workspace. Полезно скопировать этот файл поближе к основной директории, например, в папку C:\VCD и открыть в программе GTKWave. Как установить программу и как с ней работать описано во второй главе книги.

На рис. 4.19 показаны результаты трассировки.

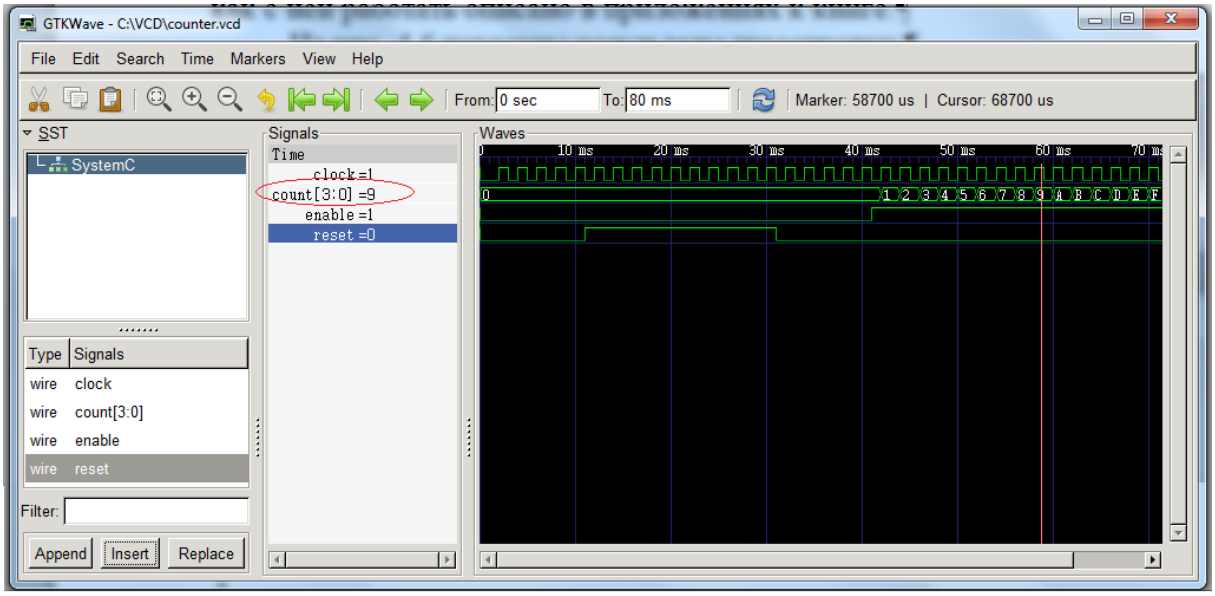


Рис. 4.19. Результаты трассировки моделирования счетчика

Установка маркера в нужной позиции показывает численные значения сигналов. Так в позиции от 58 до 60 мс выходное значение счетчика равно 9 и это значение выдается в момент 60 мс на печать.

4.12. Пример мультиплексора MUX

На рис. 4.20 показан мультиплексор MUX, который имеет два восьмибитных информационных входа `sc_in <sc_bv <bit_number> > a_in` и `sc_in <sc_bv <bit_number> > b_in`, адресный вход `sc_in <sc_bit> juht` и восьмибитный выход `sc_out <sc_bv <bit_number> > out_sig .`

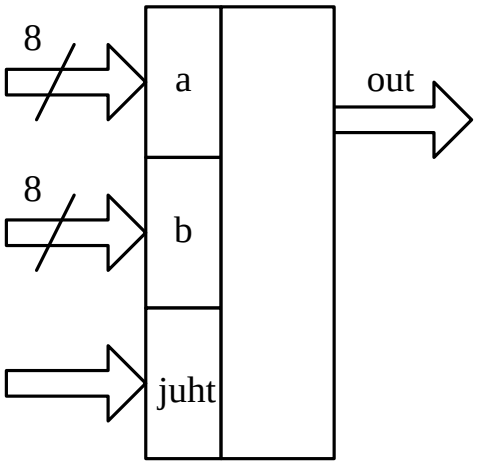


Рис. 4.20. Схема мультиплексора

В конструкторе SC_CTOR(mux) выполнена инициализация переменных, регистрация процесса SC_METHOD (`choose_out`) с чувствительностью к изменению сигналов на входах и функцией выбора выходного сигнала `void choose_out()` по адресному сигналу `juht`.

Исполняемый файл `t_mux.cpp` создает изменяющиеся стимулы на входах a, b, на адресном входе `juht` и выполняет трассировку результатов.

Листинг 4.27

Программа модели мультиплексора MUX

Заголовочный файл

```
// mux.h
#include "systemc.h"
SC_MODULE (mux) {
int const static bit_number=8;
sc_in <sc_bv <bit_number> > a_in;
sc_in <sc_bv <bit_number> > b_in;
sc_in <sc_bit> juht;
sc_out <sc_bv <bit_number> > out_sig;
SC_CTOR(mux):a_in("a_in"),b_in("b_in"),
juht("juht"),out_sig("out_sig") {
cout<<"mux constructor"<<endl;
SC_METHOD( choose_out );
sensitive<<juht<<a_in<<b_in;
}
void choose_out() {
if (juht.read())
out_sig.write(a_in.read());
else out_sig.write(b_in.read());
}
};
```

Исполняемый файл

```
// t_mux.cpp ...
void t_mux::stimulus()
{
t_a_in.write("11111111");
t_b_in.write("00001111");
```

```
    t_juht.write((sc_bit)0);
    wait(1, SC_PS);
    t_juht.write((sc_bit)1);
    wait(10, SC_NS);
    t_juht.write((sc_bit)0);
    wait(10, SC_NS);
    t_a_in.write("11110000");
    t_b_in.write("10101010");
    t_juht.write((sc_bit)0);
    wait(10, SC_NS);
    t_juht.write((sc_bit)1);
    wait();
    }
    int sc_main(int args, char* argv[])
    {
    t_mux testMux("t_mux");
    sc_trace_file *trcf=
    sc_create_vcd_trace_file("trace-it");
    if(trcf==NULL) cout<<"Sorry, no
tracing..."<<endl;
    sc_trace(trcf, testMux.t_a_in, "t_a_in");
    sc_trace(trcf, testMux.t_b_in, "t_b_in");
    sc_trace(trcf, testMux.t_juht, "t_juht");
    sc_trace(trcf, testMux.t_out_sig, "t_out_sig");
    sc_start(40, SC_NS);
    sc_close_vcd_trace_file(trcf);
    return 0;
    }
```

Результаты моделирования моделирования представлены на рис. 4.21:

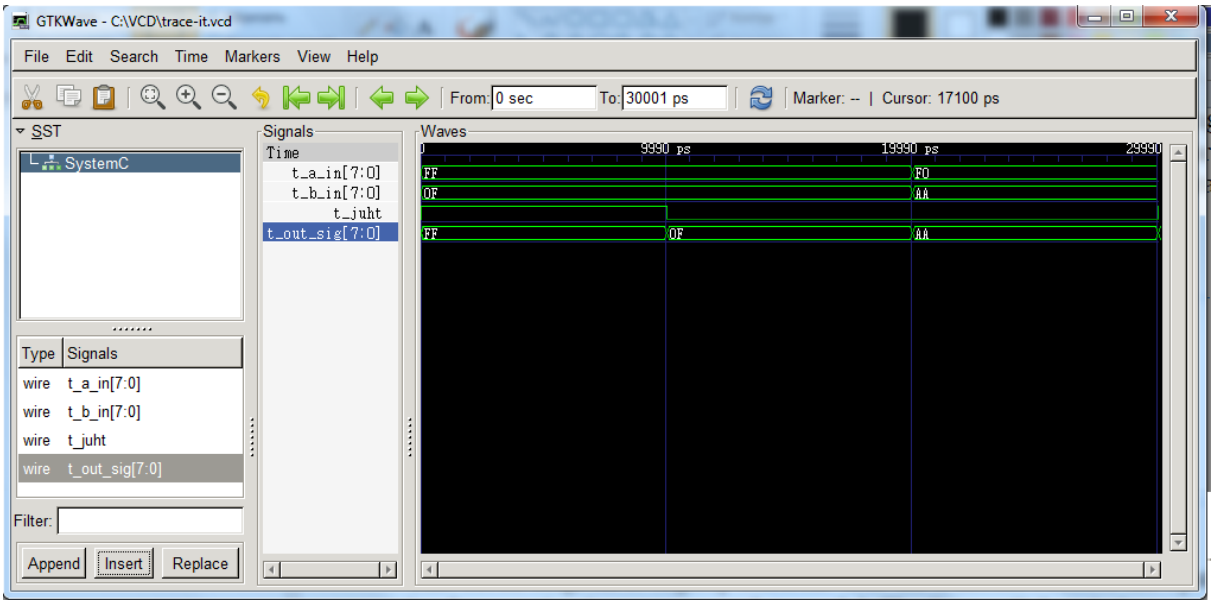


Рис. 4.21. Результаты моделирования мультиплексора

4.13. Базовый пример моделирования простой шины на уровне транзакций

Эта структура шины, показанной на рис. 4.22, является частью библиотеки примеров из SystemC-2.3.1. Шина использует общую форму синхронизации модулей, подключенных к шине, на восходящем фронте тактовых сигналов, а сама шина синхронизируется на падающем фронте такта.

Три ведущих блока (`masters`) и два ведомых (`slave`) могут быть подключены к шине. Каждый мастер идентифицируется с помощью уникального приоритета, который представлен целым числом без знака. Чем ниже этот приоритет, тем важнее мастер. Каждый ведущий связывается с шиной через интерфейс, который описывает связь между мастерами и шиной.

Возможны следующие режимы:

• **Режим блокировки**: данные перемещаются по шине в режиме пакетной съемки. Транзакция не может быть прервана запросом с более высоким приоритетом.

• **Режим без блокировки**: чтение или запись одного слова данных. После завершения транзакции, вызывающий должен позаботиться о проверке статуса последнего запроса. Статус запроса является одним из:

1. SIMPLE BUS REQUEST - простой запрос шины (запрос послан и помещен в очередь);
2. SIMPLE BUS WAIT - простое ожидание шины (запрос обслуживается, но обслуживание не завершено);

3. SIMPLE BUS OK (запрос завершен без ошибок) или
4. SIMPLE BUS ERROR (ошибка произошла во время обработки запроса).

• **Прямой режим**: функции прямого интерфейса выполняют передачу данных через шины, но без использования протокола шины. Обычно они используются для отладки состояние памяти.

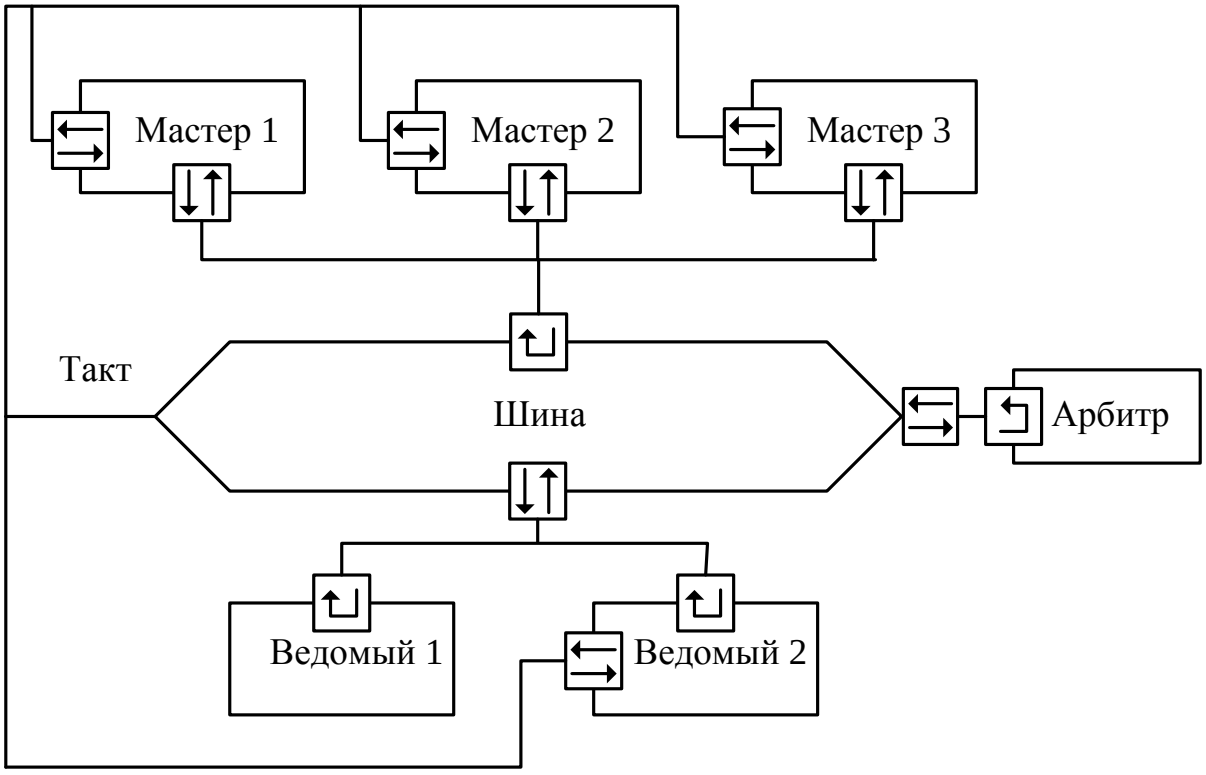


Рис. 4.22. Структура простой шины

Интерфейс ведомого устройства описывает связь между шиной и ведомыми устройствами. К шине можно подключить несколько ведомых устройств. Каждое ведомое устройство моделирует некоторую память, к которой можно получить доступ через подчиненный интерфейс. Возможны два режима:

• Прямой интерфейс: немедленное считывание или запись данных без использования протокола шины.

• Непрямой интерфейс: чтение или запись одного элемента данных, который отмечен в памяти ведомого. Функции возвращаются мгновенно, а вызывающий должен проверить статус передачи.

К шине можно подключить более одного мастера. Каждый мастер независим от других, поэтому каждый может запросить шину в любое время. Арбитр выбирает наиболее подходящий запрос в соответствии со следующими правилами:

• Если текущий запрос является заблокированным запросом пакета, он всегда выбирается.

• Если последний запрос установил флаг блокировки и снова запрошен, он будет выбран из очереди коллекции и возвращен в противном случае.

• Запрос с наивысшим приоритетом выбирается из очереди сбора и возвращается.

Обработка запросов происходит следующим образом. Запрос может вызвать разные запросы к ведомым. Это единичные запросы и они не могут быть прерваны. Ведомый может принимать несколько запросов для завершения. Каждый раз, когда запрос должен быть передан ведомому как только ведомая транзакция завершена, `m_current_request` очищается и обновляется форма запроса `(address+=4, data++)`.

Когда `m_current_request` понятен, будет выбран следующий запрос. Это может быть одно и то же, если передача не завершена (пакетный режим), или это может быть запрос с более высоким приоритетом. Возможно включение в неблокируемый пакетный режим.

Когда транзакция устанавливает блокировку, то в соответствующем в запросе установлено значение SIMPLE_BUS_LOCK_SET. Если заблокированная транзакция предоставляется арбитру первый раз, блокировка устанавливается на SIMPLE_BUS_LOCK_GRANTED.

В конце транзакции блокировка устанавливается на SIMPLE_BUS_LOCK_SET. Если теперь будет сделан новый заблокированный запрос с тем же приоритетом, устанавливается статус блокировки на SIMPLE_BUS_LOCK_GRANTED, и арбитр выберет лучшую транзакцию. Если заблокированная транзакция не была выбрана арбитром в первый раз (запрос с более высоким приоритетом превалировал), тогда блокировка не установлена. После завершения транзакции, блокировка устанавливается из SIMPLE_BUS_LOCK_SET (устанавливается во время функции Bus-interface), на SIMPLE_BUS_LOCK_NO. Шина выводится из следующих интерфейсов и содержит их реализацию:

- блокировка: `burst_read / burst_write`
- неблокирование: `read/write/get_status`
- прямая: `direct_read / direct_write`

Простая модель шины- это циклическая модель транзакционного уровня.

В «Простой модели» :

нет конвейерной обработки;

нет раздельных транзакций;

нет состояний ожидания хозяина;

нет схемы запроса / подтверждения.

Конечно, эти функции тоже можно смоделировать на уровне транзакций.

Модель использует интерфейсные вызовы методов (IMC).

Модули обмениваются данными по каналам.

Каналы реализуют интерфейсы.

Интерфейс представляет собой набор методов (функций).

Порты:

- Модули имеют порты.
- Порты подключены к каналам.
- Модули доступа к каналам через порты:

```
…..some_port-> some_method (some_data);
```

Каналы могут быть иерархическими, т. е. они могут содержать модули, процессы и каналы. Модуль, реализующий интерфейс, представляет собой иерархический канал.

Шина выполнена как иерархический канал. Арбитр и ведомые также выполнены как каналы. По желанию порты могут быть подключены к нескольким каналам. По запросам мастера получают доступ к шинам по переднему фронту синхроимпульса. Шина является процессом, чувствительным к падающему фронту. Вызывается арбитр предоставляет доступ к шине. Затем ведомые становятся доступны после просмотра карты памяти.

Полный код программы приведен в примерах SystemC-2.3.1 под названием «simple bus» и достаточно объемный. Тем не менее, весьма полезно изучить и смоделировать этот код.

Структура программы содержит заголовочные и исполняемые файлы:

Заголовочный файл simple_bus.h
Исполняемый файл simple_bus
Заголовочный файл simple_bus_arbiter
Исполняемый файл simple_bus_arbiter
Заголовочный файл simple_bus_arbiter_if_h
Заголовочный файл simple_bus_blocking_if.h
Заголовочный файл simple_bus_direct_if_h
Заголовочный файл simple_bus_fast_mem_h
Заголовочный файл simple_bus_master_blocking_h
Исполняемый файл simple_bus_master_blocking.cpp
Заголовочный файл simple_bus_master_direct.h
Исполняемый файл simple_bus_master_direct.cpp
Заголовочный файл simple_bus_master_non_blocking_h

Исполняемый файл simple_bus_master_non_blocking.cpp
Заголовочный файл simple_bus_request.h
Заголовочный файл simple_bus_slave_if.h
Заголовочный файл simple_bus_slow_mem.h
Заголовочный файл simple_bus_types.h
Исполняемый файл simple_bus_types
Заголовочный файл simple_bus_test.h
Исполняемый файл simple_bus_tools.cpp
Исполняемый файл simple_bus_main.cpp

На рис. 4.23 показаны результаты компиляции программы simple_bus в среде Eclipse.

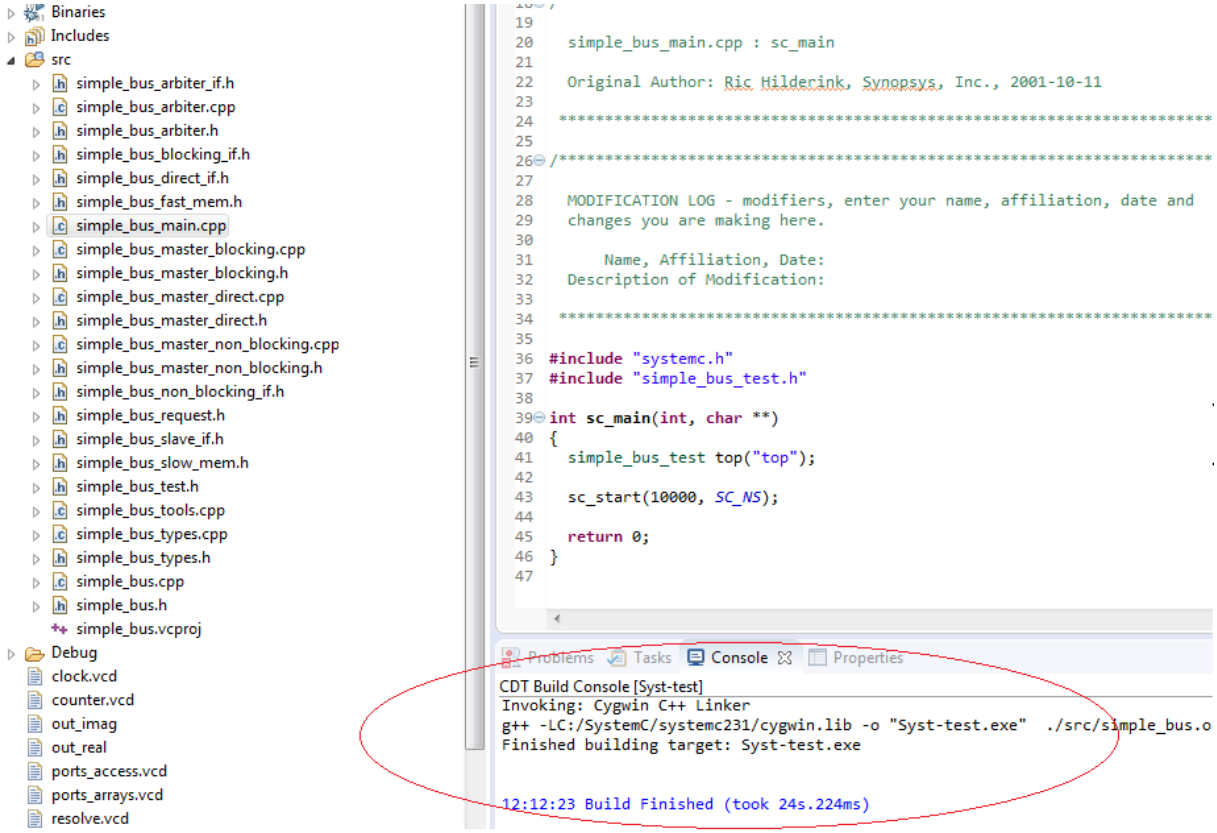


Рис. 4.23. Результат компиляции программы simple_bus

Результаты моделирования показаны на рис.4.24. С циклом 100 нс в исполняемом файле `simple_bus_master_direct.cpp` выполняется печать данных из ведомой памяти операторами:

```
sb_fprintf(stdout, "%s %s : mem[%x:%x] = (%x, %x, %x, %x)\n",
      sc_time_stamp().to_string().c_str(), name(), m_address,
      m_address+15,
```

```
mydata[0], mydata[1], mydata[2], mydata[3]);
```

```
0 s top.master_d : mem[78:87] = (0, 0, 0, 0)
100 ns top.master_d : mem[78:87] = (b, c, d, e)
200 ns top.master_d : mem[78:87] = (b, c, d, e)
300 ns top.master_d : mem[78:87] = (b, c, d, e)
400 ns top.master_d : mem[78:87] = (1b, 1d, d, e)
500 ns top.master_d : mem[78:87] = (26, 18, 1a, 2f)
600 ns top.master_d : mem[78:87] = (26, 18, 1a, 2f)
700 ns top.master_d : mem[78:87] = (26, 18, 1a, 2f)
800 ns top.master_d : mem[78:87] = (31, 24, 27, 3d)
900 ns top.master_d : mem[78:87] = (31, 24, 27, 3d)
1 us top.master_d : mem[78:87] = (31, 24, 27, 3d)
1100 ns top.master_d : mem[78:87] = (31, 24, 27, 3d)
1200 ns top.master_d : mem[78:87] = (3c, 41, 46, 5e)
1300 ns top.master_d : mem[78:87] = (3c, 41, 46, 5e)
1400 ns top.master_d : mem[78:87] = (3c, 41, 46, 5e)
1500 ns top.master_d : mem[78:87] = (47, 4d, 53, 6c)
1600 ns top.master_d : mem[78:87] = (47, 4d, 53, 6c)
1700 ns top.master_d : mem[78:87] = (47, 4d, 53, 6c)
1800 ns top.master_d : mem[78:87] = (47, 4d, 53, 6c)
1900 ns top.master_d : mem[78:87] = (62, 6a, 60, 7a)
2 us top.master_d : mem[78:87] = (62, 6a, 72, 8d)
2100 ns top.master_d : mem[78:87] = (62, 6a, 72, 8d)
2200 ns top.master_d : mem[78:87] = (62, 6a, 72, 8d)
2300 ns top.master_d : mem[78:87] = (6d, 76, 7f, 9b)
2400 ns top.master_d : mem[78:87] = (6d, 76, 7f, 9b)
2500 ns top.master_d : mem[78:87] = (6d, 76, 7f, 9b)
2600 ns top.master_d : mem[78:87] = (78, 82, 8c, a9)
2700 ns top.master_d : mem[78:87] = (88, 93, 9e, bc)
2800 ns top.master_d : mem[78:87] = (88, 93, 9e, bc)
2900 ns top.master_d : mem[78:87] = (88, 93, 9e, bc)
3 us top.master_d : mem[78:87] = (93, 9f, ab, ca)
3100 ns top.master_d : mem[78:87] = (93, 9f, ab, ca)
3200 ns top.master_d : mem[78:87] = (93, 9f, ab, ca)
3300 ns top.master_d : mem[78:87] = (9e, ab, b8, d8)
3400 ns top.master_d : mem[78:87] = (ae, ab, b8, d8)
```

Рис. 4.24. Результаты моделирования simple_bus

Аналогичные результаты были получены в среде MS Visual Studio:

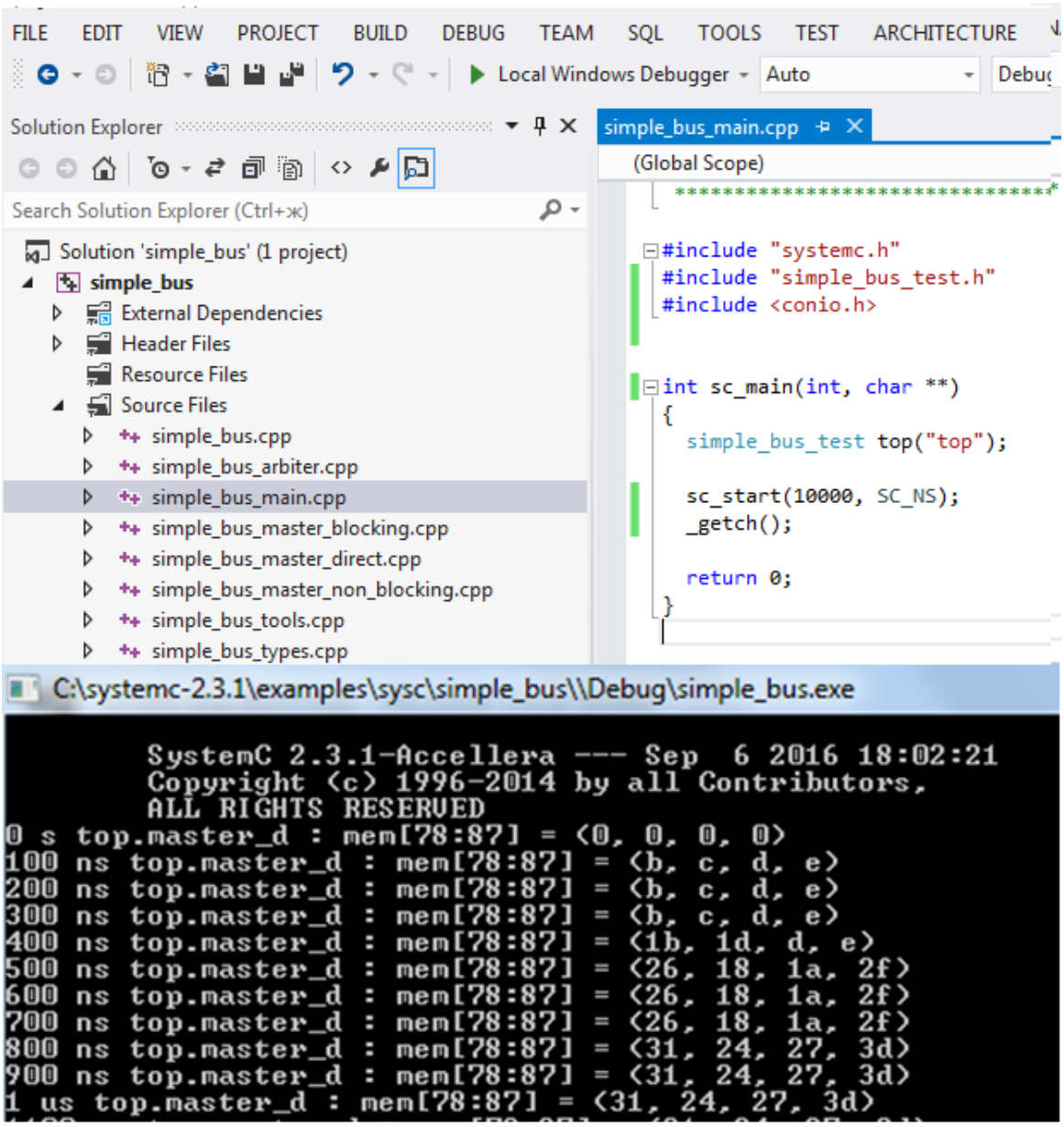


Рис. 4.25. Моделирование в среде MS Visual Studio

Глава 5. Моделирование на уровне транзакций в SystemC

В предыдуших главах мы кратко говорили о концепции моделирования, основанного на транзакциях (TLM - transaction-level modeling). В этой главе мы обсудим некоторые причины, по которым разработчик будет использовать методологию TLM, включая разработку дизайна, раннюю интеграцию аппаратного и программного обеспечения на начальних этапах разработки. Очевидно, что любая из вышеперечисленных причин является веской для разработки модели TLM. Для реализации методологии TLM должно быть точное определение - Стандарт TLM. Этот стандарт должен быть достаточно гибким, чтобы работать на разных уровнях абстракции и не зависить от протокола. В настоящее время существуют различные рабочие группы, определяющие спецификацию TLM. В этой главе мы сосредоточимся только на стандарте TLM 2.0, созданном Open SystemC Initiative (OSCI).

5.1 Введение

Стандарт моделирования уровня транзакций TLM-2.0 от Open SystemC Initiative (OSCI) был выпущен 9 июня 2008 года. Официальный релиз комплекта включает полную документацию, растущий набор примеров и может быть получен с веб-сайта OSCI, www.systemc.org. Документация в официальном релиз-комплекте достаточно хороша. Эта глава учебника дополняет документацию и поможет вам начать с представления учебного стиля для ознакомления с основными функциями TLM-2.0. В этом учебном пособии предполагается, что вы знаете SystemC и знаете основы моделирования на уровне транзакций. Знание предыдущего стандарта OSCI TLM 1.0 станет отличной отправной точкой, но не является существенным. Это руководство сопровождается набором исходных файлов из раздела Examples от Systemc-2.3.1, которые вы можете скачать. Примеры могут быть выполнены с использованием SystemC-2.3.1 и TLM-2.0, обе библиотеки доступны по адресу www.systemc.org. Примеры не могут быть запущены с TLM2.0-draft-1 или TLM2.0-draft-2, которые несовместимы с окончательной версией. Единственный другой способ – это поддержка компилятором C++. В качестве альтернативы вы можете использовать специальный симулятор SystemC, хотя вам придется платить реальные деньги за один из них.

5.2. Концепции моделирования

Моделирование уровня транзакций в SystemC включает связь между процессами SystemC с использованием вызовов функций. В центре внимания TLM лежит связь между процессами, а не алгоритмы, выполняемые

самими процессами, поэтому процессы, показанные в этой главе будет довольно тривиальным. Мы предполагаем, что в модели поведения системы некоторые процессы SystemC будут производить данные, другие будут потреблять данные, некоторые инициируют общение, иные будут пассивно реагировать на сообщение, инициированное другими. В центре внимания OSCI TLM-2.0, в частности, является моделирование встроенных микросхем с памятью. Это не означает, что TLM-2.0 выделен исключительно для карт с отображением памяти, так как именно здесь сосредоточено большинство функций. TLM-2.0 имеет многоуровневую структуру, Более низкие слои являются более гибкими и общими, а верхние слои являются специфичными для моделирования шины. В будущем стандарт может быть переориентирован к другим стилям общения по мере их появления. Очевидным направлением являются сетевые архитектуры (NoC). В качестве отправной точки мы предположим, что вы знакомы с концепциями модуля, порта, процесса, канала, интерфейса и события в SystemC. Если нет, вам следует изучить предыдущие главы. TLM-2.0 включает в себя процессы, встроенные в отдельные модули, обмениваясь сообщениями через метод интерфейса, через порты и экспорт.

5.3. Инициаторы, цели и сокеты

В TLM-2.0 инициатор - это модуль, который инициирует новые транзакции, а цель (target) - это модуль, который реагирует на транзакции, инициированные другими модулями. Транзакция представляет собой структуру данных (объект C++), переданную между инициаторами и целевыми объектами с использованием вызовов функций. Тот же модуль может действовать как в качестве инициатора, так и в качестве цели, и это, как правило, будет иметь место для модели арбитра, маршрутизатора или шины.

Для передачи транзакций между инициаторами и целевыми объектами TLM-2.0 использует сокеты (гнезда). Инициатор отправляет транзакции через сокет инициатора, а цель получает входящие транзакции через целевой сокет. Модуль, который просто пересылает транзакции без изменения их контента известен как компонент межсоединения. Компонент межсоединения будет иметь как целевой сокет, так и сокет инициатора.

Теперь давайте посмотрим на некоторый код SystemC. Модель TLM-2.0 должна включать стандартный заголовок SystemC, заголовок «`tlm.h`» и любые другие, которые вы используете в комплекте. В этом случае мы используем два сокета из каталога утилит (`tlm_utils`). Вы должны указать вложенный путь вашего компилятора C++ (или makefile) в каталоге ./include в реализации.

Листинг 5.1

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using namespace std;
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
```

При использовании симулятора OSCI необходимо определить макрос SC_INCLUDE_DYNAMIC_PROCESSES, а при использовании определенных частей набор из TLM-2.0, в частности, простые сокеты. Причина в том, что эти конкретные сокеты порождают динамические процессы. Теперь мы объявляем инициатор и целевые модули, где модуль инициатора будет генерировать транзакции, а целевой модуль будет представлять собой простую память. Трансакции, созданные инициатором, будут считываться из памяти или записываться в нее.

Листинг 5.2

```
struct Initiator: sc_module
{...};
struct Memory: sc_module
{...};
```

Мы также должны подключить иерархию модулей:

Листинг 5.3

```
SC_MODULE(Top)
{
Initiator *initiator;
Memory *memory;
SC_CTOR(Top)
{
initiator = new Initiator("initiator");
memory = new Memory ("memory");
initiator->socket.bind( memory->socket );
}
};
```

Модуль верхнего уровня иерархии создает один инициатор и одну память и связывает сокет инициатора с целевым сокетом в целевой памяти. Сокеты заключают в себе все необходимое для двусторонней связи между модулями, включая порты и экспорт для обоих направлений коммуникации. Один сокет-инициатор всегда связан с одним целевым сокетом. Со-

единитель-инициатор фактически является sc_port с sc_export на одной стороне, тогда как целевой сокет фактически является sc_export с sc_port на другой стороне. Связывающий оператор класса сокета связывает порт-экспорт в обоих направлениях с помощью одного вызова функции. Это удобное свойство функции сокетов. (Также можно связать сокеты иерархически вверх и вниз по иерархии вложенных модулей). Для полноты, при использовании симулятора OSCI вам также понадобится следующая функция sc_main:

Листинг 5.4

```
int sc_main(int argc, char* argv[])
{
Top top("top");
sc_start();
return 0;
}
```

Внутри модуля инициатора и памяти инициатор и целевые сокеты должны быть объявлены и сконструированы явно:

Листинг 5.5

```
struct Initiator: sc_module
{
tlm_utils::simple_initiator_socket<Initiator>
socket;
SC_CTOR(Initiator) : socket("socket")
{
...
};
struct Memory: sc_module
{
tlm_utils::simple_target_socket<Memory> socket;
SC_CTOR(Memory) : socket("socket")
{
...
};
```

Все объявления TLM-2.0 находятся в одном из двух пространств имен C++ `tlm` или `tlm_utils`. Чтобы было ясно, мы будем квалифицировать все такие имена явно во всех примерах. Обратите внимание на имена типов сокетов, `simple_initiator_socket < Initiator >` и `simple_target_socket <Memory>`. Первый аргумент шаблона сокета задает имя родительского модуля. Существуют другие аргументы шаб-

лона сокета, но они не показаны здесь, поскольку мы разрешаем им принимать значения по умолчанию. Другие аргументы эффективно позволяют нам задавать ширину сокета и тип транзакций, передаваемых через сокет. *Простые* инициатор и целевые сокеты так называются, потому что они просты в использовании. Это классы утилит, полученные из двух базовых типов сокетов: `tlm_initiator_socket` и `tlm_target_socket`. Вам не нужно знать об этих последних двух базовых классах, если вы не хотите создавать Ваши собственные удобные сокеты, что является полезным, но более совершенным методом кодирования. Пока мы будем придерживаться простых сокетов, которые позволяет инициатору и цели относительно легко кодировать транзакции. Строго говоря, основные базовые классы являются ключевыми для интероперабельности, а не служебными сокетами. Классы в пространстве имен `tlm_utils` существуют для удобства и производительности. Только классы в *пространстве имен* `tlm` крайне важны для взаимодействия. В этом примере инициатор будет связываться с целевой памятью с помощью транспортного интерфейса блокировки, поэтому целевой стороне необходимо реализовать единственный метод с именем `b_transport.` При использовании простого целевого сокета это делается путем установления целевым регистром метода обратного вызова с разъема следующим образом:

```
    socket.register_b_transport(this,
&Memory::b_transport);
```

Теперь все, что нужно сделать - это обеспечить реализацию метода `b_transport`, который мы опишем ниже.

5.4. Общая полезная нагрузка и блокирующий транспорт

Тип транзакции по умолчанию для классов сокетов, подразумеваемый в отсутствие любых аргументов шаблона, - `tlm_generic_payload`. Общая полезная нагрузка является важной частью стандарта TLM-2.0, поскольку она является еще одним из ключей к достижению взаимодействия между уровнем транзакции моделей.

Общая полезная нагрузка включает в себя: команду, адрес, данные, используемые байты, статус ответа, расширения.

Что такое полезная нагрузка?

Это простой структурированный объект, содержащий:

- Управление (команда);
- Данные;
- Различные свойства и расширения;
- Предназначен для шин с памятью;
- Размер пакета;
- Ширина байта;

• Определена как класс для гибкости;

• Полезная нагрузка передается по ссылке для ускорение вызовов путем сокращения времени на копирование;

• Необходимо знать время жизни.

Общая полезная нагрузка служит двум близким целям. Она может использоваться как тип транзакции общего назначения для абстрактной памяти при моделировании шины, когда вас не интересуют точные детали любого конкретного протокола шины, предполагая немедленную функциональную совместимость между моделями. Альтернативно, общая полезная нагрузка может использоваться в качестве основы для моделирования широкого спектра конкретных протоколов на более подробном уровне. Красота этого подхода заключается в том, что относительно легко соединяться между различными протоколами, когда оба построены поверх того же общего типа полезной нагрузки. Наш модуль-инициатор имеет `thread process` для генерации потока общих транзакций полезной нагрузки.

Листинг 5.6

```
    SC_CTOR(Initiator) : socket("socket")
    {
    SC_THREAD(thread_process);
    }
    void thread_process()
    {
    tlm::tlm_generic_payload* trans = new
tlm::tlm_generic_payload;
    sc_time delay = sc_time(10, SC_NS);
    for (int i = 32; i < 96; i += 4)
    {
    ...
    socket->b_transport( *trans, delay );
    ...
    }
    }
```

Транзакция отправляется через сокет, используя метод `b_transport` транспортного интерфейса блокировки TLM-2.0, который передает его аргумент транзакции по ссылке и не имеет возвращаемого значения. Инициатор несет ответственность за выделение и удаление хранилища для транзакции.

Вызов `b_transport` также содержит временную аннотацию, которая должна быть добавлена к текущему времени моделирования (как возвращение `sc_time_stamp`), чтобы определить время обработки транзак-

ции. Временные аннотации синхронизации активны как для вызова, так и для возврата из b_transport.

Листинг 5.7

```
    tlm::tlm_command cmd = static_cast(rand() % 2);
    if (cmd == tlm::TLM_WRITE_COMMAND) data =
0xFF000000 | i;
    trans->set_command( cmd );
    trans->set_address( i );
    trans->set_data_ptr( reinterpret_cast<unsigned
char*>(&data) );
    trans->set_data_length( 4 );
    trans->set_streaming_width( 4 );
    trans->set_byte_enable_ptr( 0 );
    trans->set_dmi_allowed( false );
    trans->set_response_status(
tlm::TLM_INCOMPLETE_RESPONSE );
    socket->b_transport( *trans, delay );
```

Каждая общая транзакция полезной нагрузки имеет стандартный набор атрибутов шины: команда, адрес, данные, байт, ширина потока и ответный статус. Общая полезная нагрузка также содержит подсказку и расширения DMI. Хотя каждый атрибут имеет значение по умолчанию, рекомендуется использовать не менее 8 из 10 атрибутов явно перед передачей транзакции на вызов метода интерфейса. Причина в том, что объекты транзакции обычно используются повторно из пула.

Общая полезная нагрузка поддерживает две команды: чтение и запись. Здесь атрибут команды настроен на чтение или запись произвольным образом. Атрибут адреса - это самое низкое значение адреса, по которому данные должны быть прочитаны или записаны. Здесь адрес устанавливается в индекс цикла. Атрибут указателя данных указывает на буфер данных внутри инициатора, а атрибут длины данных дает длину массива данных в байтах. Здесь длина данных равна 4 байтам. В случае команды записи данные будут скопированы из массива данных в целевой объект, а в случае команды чтения, скопированны с целевого объекта в массив данных. В любом случае фактическая копия выполняется в цели. Атрибут ширины потоковой передачи определяет ширину потокового пакета, где адрес повторяется. Для транзакции без потоковой передачи ширина потока должна равняться длине данных, как это имеет место здесь. Хотя значение по умолчанию для атрибута ширины потоковой передачи равно 0, значение 0 не разрешается, когда транзакция приходит к отправке через вызов метода интерфейса. Этот же принцип применяется к указателю данных и данным длины.

Указатель включения байта установлен в 0, чтобы указать, что байтовые включения не используются. Существует также атрибут байтовой длины, который здесь не задан, потому что с указателем, установленным в 0, он будет проигнорирован. Метод `set_dmi_allowed` устанавливает подсказку DMI, которая всегда должна быть инициализирована на `false`. Атрибут подсказки DMI может быть установлен объектом, чтобы указывать, что он поддерживает интерфейс прямой памяти. Статус ответа всегда должен быть инициализирован значением `TLM_INCOMPLETE_RESPONSE`. Статус ответа может быть задан целью. Десятый общий атрибут полезной нагрузки, не упомянутый выше, представляет собой массив расширений. По умолчанию любые расширения могут быть проигнорированы инициатором или целью. Способ блокирования транспорта реализуется в целевой памяти. Во-первых, набор из шести атрибутов, которые нельзя игнорировать, извлекается из общей полезной нагрузки транзакции. (Остальные атрибуты - это подсказка DMI и статус ответа, которые задаются целевым объектом, длина разрешения байта, которые можно игнорировать, если байтовые включения не используются, и расширения, которые в любом случае можно игнорировать.)

Листинг 5.8

```
    virtual void b_transport(
tlm::tlm_generic_payload& trans, sc_time& delay )
    {
    tlm::tlm_command cmd = trans.get_command();
    sc_dt::uint64 adr = trans.get_address() / 4;
    unsigned char* ptr = trans.get_data_ptr();
    unsigned int len = trans.get_data_length();
    unsigned char* byt = trans.get_byte_enable_ptr();
    unsigned int wid = trans.get_streaming_width();
```

Затем атрибуты проверяются, чтобы гарантировать, что инициатор не пытается использовать функции, которые не может поддерживать целевой объект. В этом случае цель - память не поддерживает байтовые включения, ширину потоковой передачи или пакетные передачи. Следующее утверждение также проверяет, что адрес не выходит за пределы. Если транзакция не может быть выполнена, обработчик отчета SystemC вызывается для генерации ошибки.

Листинг 5.9

```
    if (adr >= sc_dt::uint64(SIZE) || byt != 0 || len
> 4 || wid < len)
    SC_REPORT_ERROR("TLM-2",
```

```
    "Target does not support given generic payload
transaction");
```

Конечно, если цель не может поддерживать определенные функции общей полезной нагрузки, это ограничит совместимость, но, по крайней мере, наборы функций четко определены и существуют стандартные обязательства по проверке и сообщению о несовместимости. Затем цель реализует команду чтения и записи путем копирования данных в область или из области данных в инициаторе. Что касается суждения, то правило состоит в том, что общая полезная нагрузка принимает тот же смысл, что и главный компьютер. Пока целевая память также моделируется с использованием хоста `Endianness`, копирование данных может быть выполнено с помощью `memcpy`:

Листинг 5.10

```
    if ( cmd == tlm::TLM_READ_COMMAND )
    memcpy(ptr, &mem[adr], len);
    else if ( cmd == tlm::TLM_WRITE_COMMAND )
    memcpy(&mem[adr], ptr, len);
```

Последним действием метода блокирующего транспорта является установка атрибута статуса ответа общей полезной нагрузки для указания успешного завершения транзакции. Если не задано, статус ответа по умолчанию указывает инициатору, что транзакция неполна.

```
    trans.set_response_status( tlm::TLM_OK_RESPONSE
);
```

5.5. Временная аннотация

Поскольку метод блокирующего транспорта только моделирует функциональность цели, а не моделирует любую деталь времени, он просто игнорирует значение аргумента задержки и возвращает его инициатору нетронутым.

После вызова `b_transport` инициатор проверяет статус ответа:

```
    if (trans->is_response_error() )
    SC_REPORT_ERROR("TLM-2", "Response error from
b_transport");
```

Инициатору теперь нужно реализовать любые аннотированные сроки. Поскольку эта модель касается только функциональности, она может продолжать накапливать задержки до бесконечности. Идея заключается в том, чтобы имитационная модель вычислила функциональность инициатора и цели на полной скорости и отслеживала время, затрачиваемое на лю-

бые смоделированные ресурсы, просто увеличивая переменную «сбоку». Такой стиль кодирования называется слабоуровневым в TLM-2.0. Однако то, что на самом деле делает модель, - это просто ожидание указанной задержки при возврате из вызова `b_transport`. Это будет замедлять симуляцию, поскольку для транзакции требуется контекстный переключатель. Но этого достаточно для простого примера и делает журнал моделирования простым для интерпретирования.

5.6. Взаимодействие и базовый протокол

Таким образом TLM-2.0 обеспечивает совместимость между моделями, использующими стандартный набор API, предоставляет дополнительные классы полезности для улучшения производительности и поощряет последовательный стиль кодирования.

Ключами к совместимости в TLM-2.0 являются:

- использование стандартного инициатора и целевых сокетов, один из которых должен быть создан для каждого соединения с шиной памяти или другой коммуникационным ресурсом;
- использование общей полезной нагрузки, которая должна быть создана и задана для представления атрибутов каждого объекта транзакции;
- использование базового протокола.

Нам не нужно было изучать детали базового протокола в этой главе учебника, но это подразумевается при использовании простых сокетов и метода `b_transport`. Базовый протокол определяет правила использования общей полезной нагрузки со стандартными интерфейсами TLM-2.0 и сокетами. Блокирующий транспортный интерфейс должен использоваться везде, где транзакция может быть выполнена в одном вызове функции. Это эффективно, когда запрос транзакции переносится вызовом `b_transport`, и ответ передается с возвратом из `b_transport`. В этом примере один объект транзакции повторно используется через вызовы. Хранилище для объекта транзакции назначается инициатором в начале и после запуска. Это приемлемо, поскольку только одна транзакция «в полете» в любой момент времени. Управление памятью тривиально и обрабатывается инициатором.

Еще одна особенность этого конкретного примера заключается в том, что метод блокирования транспорта не используется, то есть не вызывает `wait.b_transport` , но может вызвать `wait.`Однако, и в принципе мы могли бы иметь ситуацию, когда есть несколько одновременных вызовов `b_transport` через один и тот же сокет от нескольких потоков в инициаторе, возможно, с противоречивыми аннотациями времени. Такая ситуация будет разрешена в соответствии с правилами базового протокола.

Блокирующий транспортный интерфейс предназначен для поддержки слабоограниченного стиля кодирования, где основное внимание уделяется функциональному исполнению с минимальной детализация времени и минимальными накладными расходами на моделирование.

5.7. Интерфейсы TLM-2.0

TLM-2.0 состоит из набора основных интерфейсов, глобальных частей, инициаторных и целевых сокетов, общей полезной нагрузки, базового протокола и утилит. Также включены основные интерфейсы TLM-1, интерфейс анализа и порты анализа, хотя они отделены от основного тела стандарта TLM-2.0. Основные интерфейсы TLM-2.0 состоят из блокирующих и неблокирующих транспортных интерфейсов, интерфейса прямой памяти (DMI) и отладки транспортного интерфейса. Общая полезная нагрузка поддерживает абстрактное моделирование подключенных к памяти шин вместе с механизмом расширения для поддержки моделирования конкретных протоколов шины, обеспечивая максимальную совместимость.

Классы TLM-2.0 накладываются поверх библиотеки классов SystemC, как показано на диаграмме (рис. 5.1). Для максимальной совместимости и, в частности, для моделирования шины с памятью рекомендуется, чтобы основные интерфейсы TLM-2.0, сокеты, общая полезная нагрузка и базовый протокол были вместе совместно использованы. Эти классы известны в совокупности как уровень взаимодействия. В случаях, когда общая полезная нагрузка неприемлема, основные интерфейсы, инициатор и целевые сокеты или основной интерфейс сам по себе могут использоваться с альтернативным типом транзакции. Даже технически возможно, что общая полезная нагрузка будет использоваться непосредственно с основными интерфейсами без инициатора и целевых сокетов, хотя этот подход не рекомендуется. Не обязательно использовать утилиты для обеспечения совместимости между моделями шин. Тем не менее, эти классы должны использоваться там, где это возможно, для согласованности стиля и документируются и поддерживаются как часть стандарта TLM-2.0.

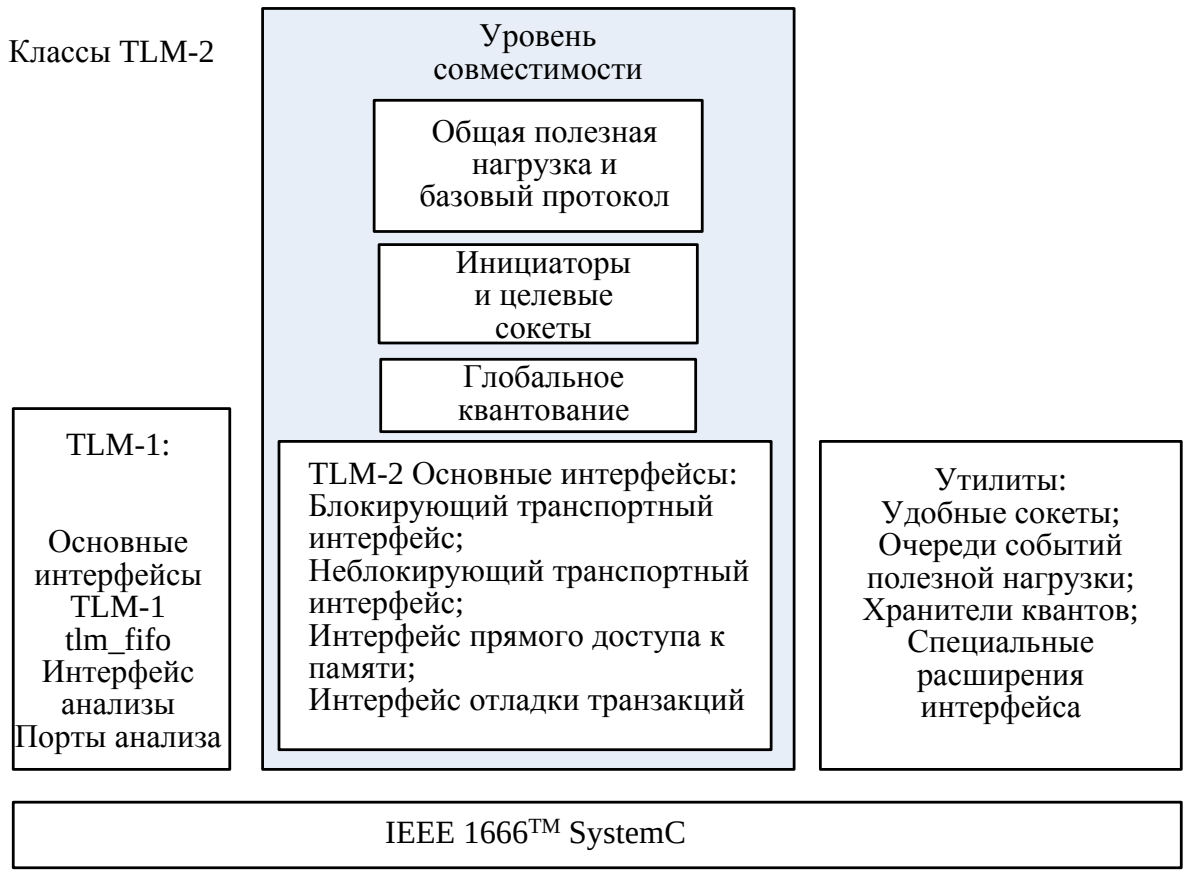


Рис. 5.1. Классы TLM 2.0

Общая полезная нагрузка в основном предназначена для моделирования шины с отображением памяти, но может также использоваться для моделирования других протоколов без шины с аналогичными атрибутами. Атрибуты и этапы общей полезной нагрузки могут быть расширены для моделирования конкретных протоколов, но такие расширения могут привести к сокращению функциональной совместимости в зависимости от степени отклонения от стандартной не расширенной общей полезной нагрузки. Ожидается, что быстрая, слабозатухающая модель использует блокирующий транспортный интерфейс, интерфейс прямой памяти и временную развязку. Ожидается, что более точная, ориентированная по времени модель будет использовать неблокирующий транспортный интерфейс и очереди событий полезной нагрузки. Однако эти утверждения представляют собой только предложения стиля кодирования и не являются нормативной частью стандарта TLM-2.0.

Далее мы рассматриваем применение документа OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL. Software version: TLM 2.0.1 Document version: JA32. Этот документ является окончательным справочным руководством по стандарту TLM-2.0. Основное внимание в этом документе уделяется ключевым концепциям и семантике классов TLM-2.0,

включая утилиты. Он не описывает все вспомогательные коды, примеры и модульные тесты. В нем перечислены основные интерфейсы TLM-1, но они не определяют их семантику.

5.8. Исходный код и документация

Версия TLM-2.0 имеет иерархическую структуру каталогов:

`include/tlm` - исходный код C++ стандарта TLM-2.0 с файлами `readme` и примечаниями к выпуску;

`./tlm_h` – интерфейс взаимодействия TLM-2.0;

`./tlm_h/tlm_2`- основные интерфейсы TLM-2;

`./tlm_h/tlm_generic_payload` – общая полезная нагрузка TLM-2 ;

`./tlm_h/tlm_sockets` – сокеты TLM-2;

`./tlm_h/tlm_quantum` – глобальные части TLM-2;

`./tlm_1` - TLM-1 и анализ;

`./tlm_1/tlm_req_rsp` – стандарт TLM-1;

`./tlm_1/tlm_req_rsp/tlm_1_interfaces` – основные интерфейсы TLM-1;

`./tlm_1/tlm_req_rsp/tlm_channels` - TLM-1 `fifo` и `req-rsp` каналы;

`./tlm_1/tlm_req_rsp/tlm_ports` - TLM-1 неблокирующие порты с поиском событий;

`./tlm_1/tlm_req_rsp/tlm_adapters` - TLM-1 совместимые адаптеры и адаптеры для переноса;

`./tlm_1/tlm_analysis` – интерфейс анализа и порты;

`./tlm_utils` – стандартные классы утилит TLM-2, не являются существенными для совместимости.

`docs` – документация, включающая User Manual, документы и примеры;

`examples`- набор примеров, ориентированных на приложения с их описаниями;

`unit_test` – набор регрессивных тестов.

5.9. Моделирование на уровне транзакций, варианты использования и абстракция

Стандарт TLM-2.0 определяет набор интерфейсов, которые следует рассматривать как низкоуровневые механизмы программирования для реализации моделей уровня транзакций, а затем описывает ряд стилей коди-

рования, которые подходят для различных вариантов использования, но не привязаны к ним.

Определения стандартных интерфейсов TLM-2.0 отличаются от описаний стилей кодирования. Это интерфейсы TLM-2.0, которые составляют нормативную часть стандарта и обеспечивают совместимость. Каждый стиль кодирования может поддерживать диапазон абстракции по функциональности, времени и связи. В принципе пользователи могут создавать свои собственные стили кодирования.

Вневременная функциональная модель, состоящая из одного программного потока, может быть записана как функция C или как один процесс SystemC и иногда называется алгоритмической моделью. Такая модель не является уровнем транзакции как таковой, поскольку по определению транзакция является абстракцией связи, а однопоточная модель не имеет межпроцессного взаимодействия. Модель уровня транзакции требует нескольких процессов SystemC для имитации одновременного выполнения и связи.

Абстрактная модель уровня транзакции, содержащая несколько процессов (несколько программных потоков), требует некоторого механизма, с помощью которого эти потоки могут обеспечивать контроль над собой. Это связано с тем, что SystemC использует совместную многозадачную модель, в которой процесс выполнения не может быть предотвращен каким-либо другим процессом. Процессы SystemC контролируют выход, вызывая `wait` в случае *процесса потока* или возвращаясь к ядру в случае *процесса метода*. Вызовы ожидания обычно скрываются за интерфейсом программирования (API), который может моделировать специфический абстрактный или конкретный протокол, который может или не может опираться на информацию о времени.

Синхронизация может быть сильной в том смысле, что последовательность событий связи точно определяется заранее, или слабой в том смысле, что последовательность событий связи частично определяется подробным сроком отдельных процессов. Сильная синхронизация легко внедряется в SystemC с использованием FIFO или семафоров, что позволяет полностью исключить стиль моделирования, где в принципе симуляция может работать без увеличения времени моделирования. В этом смысле вневременное `(untimed)` моделирование выходит за рамки TLM-2.0. С другой стороны, быстрая виртуальная платформа, позволяющая параллельно запускать несколько встроенных программных потоков, может использовать сильную или слабую синхронизацию. В этом стандарте подходящий стиль кодирования для такой модели называется слабо синхронизированным (*loosely-timed*.)

Для более подробной модели уровня транзакции может потребоваться связать несколько временных точек протокола с каждой транзакцией, например, точки синхронизации, чтобы отметить начало и конец каждой фазы протокола. Выбирая подходящее количество точек синхронизации, можно моделировать связь с высокой степенью точности синхронизации без необходимости выполнять модели компонентов в каждом отдельном такте. В этом стандарте такой стиль кодирования называется приблизительным (*approximately-timed*.).

5.10. Стили кодирования

Стиль кодирования - это набор идиом языка программирования, которые хорошо работают вместе, а не конкретный уровень абстракции или интерфейс программного обеспечения. Для простоты и ясности мы ограничимся изучением двух конкретных названных стилей кодирования: *loosely-timed* and *approximately-timed*. По своей природе стили кодирования точно не определены, и правила, определяющие основные интерфейсы TLM-2.0, определяются независимо от этих стилей кодирования. В принципе, можно было бы определить другие стили кодирования на основе механизмов TLM-1 и TLM-2.0.

На рис. 5.2 показаны различные задачи и случаи использования стилей кодирования и применяемых механизмов.

5.10.1. Стиль untimed

TLM-2.0 не дает четкого представления о стиле несвязанного кодирования, поскольку для всех современных систем на основе шины требуется некоторое понятие времени для моделирования программного обеспечения, работающего на одном или нескольких встроенных процессорах. Однако несвязанное моделирование поддерживается основными интерфейсами TLM-1. Термин `untimed` иногда используется для обозначения моделей, которые содержат ограниченную информацию о времени неопределенной точности. В TLM-2.0 такие модели будут называться временными.

TLM 2.0 вводит два термина:

• Свободное временное (LT) моделирование - «быстрое»

• Приближенное временное (AT) моделирование – «точное».

Терминология, которая грубо определяет:

• Скорость моделирования;

• Точность синхронизации;

• Применимость модели;

• Устаревшую терминология - Untimed (отключенная модель), Programmers View with Timing (PVT) -просмотр программистов с таймин-

гом для определения задержки между поступлением команды и ее исполнением и др.

5.10.2. Стиль Loosely-timed и временная развязка

В свободно-временном стиле кодирования используется блокирующий транспортный интерфейс. Этот интерфейс позволяет связать только две точки синхронизации с каждой транзакцией, соответствующей вызову и возврату из функции блокирования транспорта. В случае базового протокола первая точка синхронизации отмечает начало запроса, а вторая знаменует начало ответа. Эти два момента времени могут возникать при одном и том же времени моделирования или в разное время.

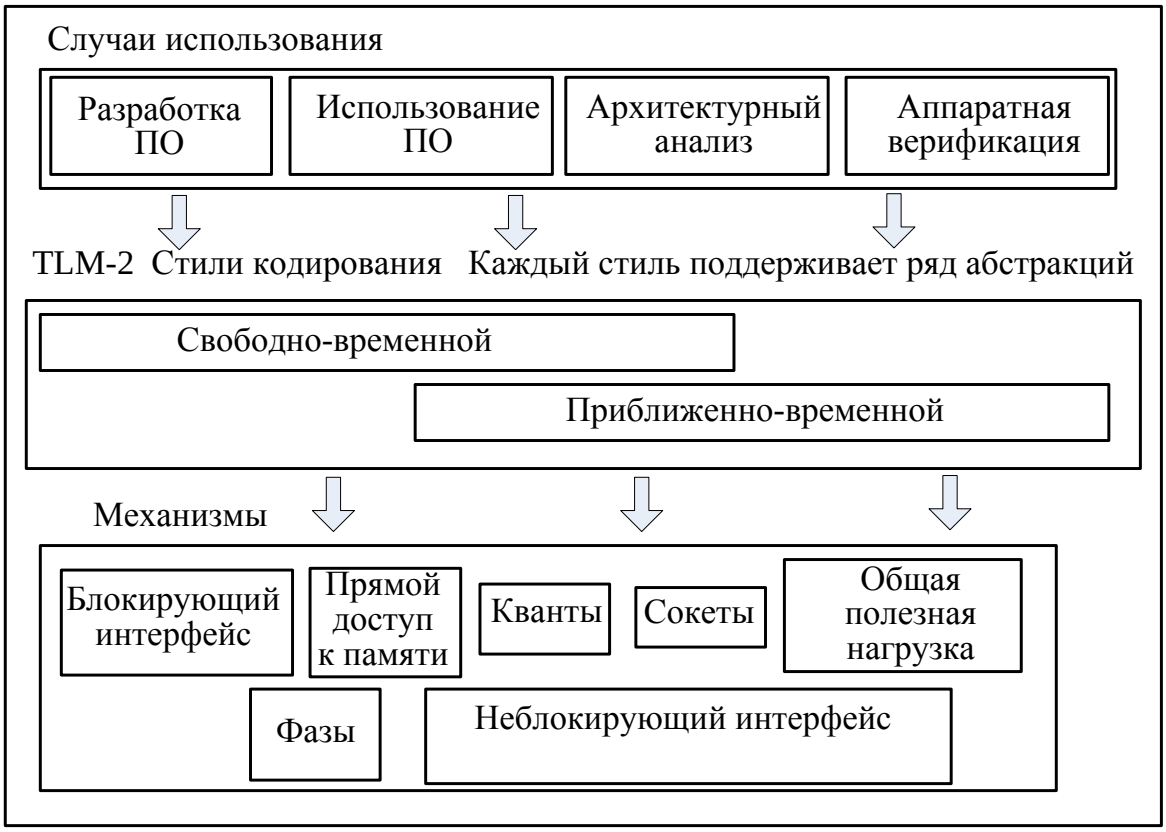


Рис. 5.2. Задачи, стили кодирования и механизмы моделей

Свободно-временный стиль кодирования подходит для использования в разработке программного обеспечения с использованием модели виртуальной платформы MPSoC, где содержание программного обеспечения может включать одну или несколько операционных систем. Свободно-временный стиль кодирования поддерживает моделирование таймеров и прерываний, достаточное для загрузки операционной системы и запуска произвольного кода на целевой машине.

Свободно-синхронизированный стиль кодирования также поддерживает временную развязку, когда отдельные процессы SystemC разрешены

для запуска в локальном временном режиме без фактического увеличения времени моделирования до тех пор, пока они не достигнут точки, когда им необходимо синхронизировать с остальной системой. Временная развязка может привести к очень быстрой симуляции для определенных систем, поскольку она увеличивает местоположение данных и кода и снижает затраты на планирование симулятора. Каждому процессу разрешается запускать определенный фрагмент времени или квант, прежде чем переключиться на следующий, или вместо этого процесс может получить контроль, когда он достигнет явной точки синхронизации.

Если просто рассматривать SystemC, то планировщик SystemC не задерживает время моделирования. Планировщик ускоряет время моделирования до момента следующего события, затем запускает любые процессы из-за запуска в это время или по событию, чувствительному для процесса. Процессы в SystemC выполняются только в текущее время моделирования (полученное вызовом метода `sc_time_stamp`), и всякий раз, когда процесс SystemC читает или записывает переменную, он обращается к состоянию переменной, как это было бы при текущем времени моделирования. Когда процесс завершается, он должен передать управление обратно в ядро моделирования. Если имитационная модель написана на мелкодетальном уровне, то дополнительные расходы на планирование событий и переключение контекста процесса становятся доминирующим фактором в скорости моделирования. Один из способов ускорить моделирование - позволить процессам работать впереди текущего времени моделирования или временной развязки.

При реализации временной развязки в SystemC процесс может быть запущен до времени моделирования, пока ему не потребуется взаимодействовать с другим процессом, например, чтобы прочитать или обновить переменную, принадлежащую другому процессу. В этот момент процесс может либо получить доступ к текущему значению, либо продолжиться (с некоторой возможной потерей точности синхронизации) или может вернуть управление ядру моделирования, только возобновив процесс, когда время моделирования затягивается с локальной временной деформацией. Каждый процесс отвечает за определение того, может ли он работать до времени моделирования без нарушения функциональности модели. Когда процесс встречается с внешней зависимостью, он имеет два варианта: либо принудительную синхронизацию, что означает предоставление возможности запускам всех других процессов в нормальном режиме до тех пор, пока время моделирования не достигнет конца. Второй вариант - не проверить возможность, обновить текущие значения и продолжить процесс.

Если бы процесс был разрешен для запуска до времени моделирования без ограничений, планировщик SystemC не смог бы работать, а другие процессы никогда не получили бы возможности выполнения. Этого можно

избежать с помощью ссылки на глобальный квант, который накладывает верхний предел на время, когда процессу разрешено работать до окончания времени моделирования. Квант устанавливается приложением, а квантовое значение представляет собой компромисс между скоростью и точностью моделирования. Слишком маленькие квантовые шаги выдают синхронизацию очень часто, но замедляют моделирование. Слишком большой квант может ввести временные несоответствия в системе, возможно, до такой степени, когда система перестает функционировать.

Например, рассмотрим моделирование системы, состоящей из процессора, памяти, таймера и некоторых медленных внешних периферийных устройств. Программное обеспечение, выполняемое на процессоре, тратит большую часть времени на получение и выполнение инструкций из системной памяти и взаимодействует только с остальной частью системы, когда она прерывается таймером, скажем, каждые 1 мс. Симулятор набора инструкций ISS, который моделирует процессор, может быть разрешен для запуска до времени моделирования SystemC с квантом до 1 мс, что обеспечивает прямой доступ к модели памяти, но только для синхронизации с периферийными моделями с частотой прерываний таймера. Дело здесь в том, что ISS не нужно запирать на время моделирования аппаратной части системы, как это было бы при более традиционном совместном симуляторе аппаратного обеспечения. В зависимости от детализации моделей, временная развязка сама по себе может дать улучшение скорости моделирования примерно 10X или 100X в сочетании с DMI.

В TLM-2.0 временная развязка поддерживается классом `tlm_global_quantum` и аннотацией времени блокирующего и неблокирующего транспортного интерфейса. Класс утилиты `tlm_quantumkeeper` обеспечивает удобный способ доступа к глобальному кванту.

5.10.3. Характеристика свободно-временных и приближенно-временных стилей кодирования

Стили кодирования можно охарактеризовать в терминах временных точек и временной развязки.

Свободно - временные. Каждая транзакция имеет только два момента времени, обозначая начало и конец транзакции. Время моделирования используется, но процессы могут быть временно отключены от времени моделирования. Каждый процесс ведет подсчет того, как далеко он прошел впереди времени моделирования, и может появиться, потому что он достигает явной точки синхронизации или потому, что он потребляет квант времени.

Приближенно-временные. Каждая транзакция имеет несколько временных точек. Обычно процессы должны запускаться в режиме блоки-

ровки с временем моделирования SystemC. Задержки, аннотированные для взаимодействия процессов, реализуются с использованием тайм-аутов (ожидание) или уведомлений о временном событии.

Untimed. Понятие времени моделирования не требуется. Производятся процессы при явных предопределенных точках синхронизации.

5.10.4. Переключение между свободно-временным и приближенно- временным моделированием.

Модель может переключаться между свободно-временным и приближенно-временным стилем кодирования во время моделирования. Идея состоит в том, чтобы быстро запускать последовательность сброса и загрузки на свободно-временном уровне, а затем переключиться на приближенно временное моделирование для более подробного анализа, как только симуляция достигнет интересной стадии.

В приведенной ниже таблице (рис. 5.3) представлено сопоставление между вариантами использования для моделирования уровня транзакций и стилей кодирования:

Случаи использования	**Стиль кодирования**
Разработка программных приложений	Свободно-временной
Анализ производительности ПО	Свободно-временной
Архитектурный анализ аппаратных средств	Свободно-временной или приближенно-временной
Проверка производительности аппаратных средст	Приближенно-временной или с точностью до циклов
Функциональная проверка аппаратных средств	Невременной (среда проверки), свободно-временной или приближенно-временной

Рис. 5.3. Сопоставление вариантов задач и кодирования

5.11. Мосты транзакций

Интерфейсы ядра TLM-2.0 передают транзакции между инициаторами и целями. Напомним, что инициатор - это модуль, который может инициировать транзакции, т.е. создавать новые объекты транзакций и передавать их, вызывая метод одного из основных интерфейсов. Цель - это мо-

дуль, который выступает в качестве конечного пункта назначения для транзакции. В случае записи транзакции, инициатор (например, процессор) записывает данные в цель (например, память). В случае транзакции чтения инициатор считывает данные из целевого объекта. Компонент межсоединений - это модуль, который обращается к транзакции, но не выступает в качестве инициатора или цели для этой транзакции, типичными примерами являются арбитры и маршрутизаторы. Роли инициатора, межсоединения и цели могут динамически меняться. Например, данный компонент может выступать в качестве межсоединения для некоторых транзакций, но как цель для других транзакций.

Чтобы проиллюстрировать эту идею, будет описано время жизни типичного объекта транзакции. Объект транзакции создается инициатором и передается в качестве аргумента метода транспортного интерфейса (блокирующего или неблокирующего). Этот метод реализуется компонентом межсоединения, таким как арбитр, который может считывать атрибуты объекта транзакции, прежде чем передавать его на другой транспортный вызов. Второй транспортный метод реализуется вторым компонентом межсоединений, таким как маршрутизатор, который, в свою очередь, передает транзакцию через третий транспортный вызов к цели, такой как память, конечный пункт назначения для объекта транзакции. (Фактическое количество компонентов межсоединений будет варьироваться от транзакции к транзакции. Их может и не быть.) Эта последовательность вызовов методов известна как прямой путь. Транзакция выполняется в целевом объекте, и объект транзакции может быть возвращен инициатору одним из двух способов: либо переносом с возвратом из вызовов метода транспорта, когда они разветвляются (это называются обратным путем), либо передаются путем явной транспортной передачи. Метод вызывает противоположный путь от цели до инициатора, известный как обратный путь. Этот выбор определяется возвращаемым значением из неблокирующего транспортного метода. (Строго говоря, есть два пути возврата, соответствующие прямым и обратным путям, но смысл обычно ясен из контекста.)

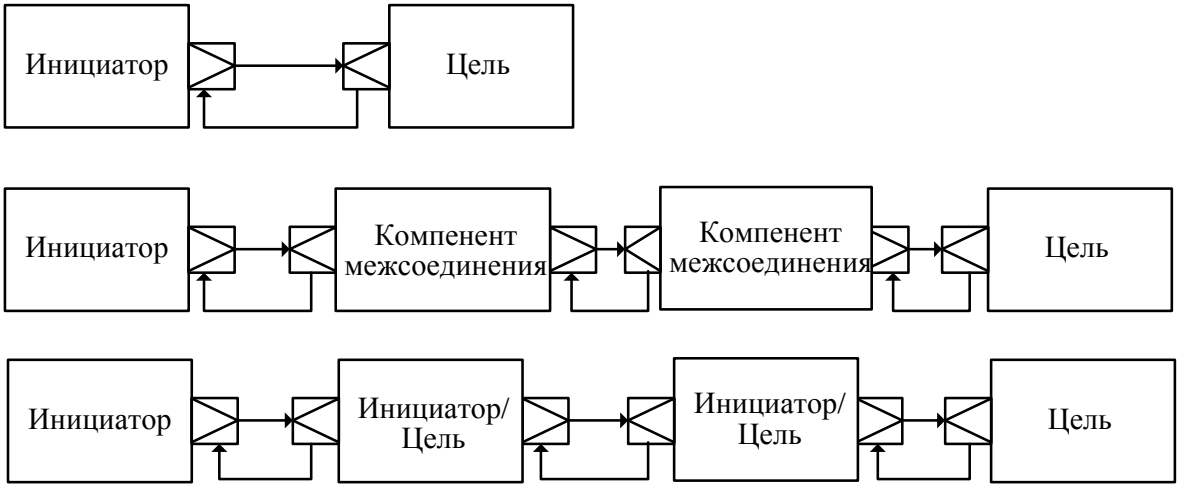


Рис. 5.4. Различные варианты соединений

Прямой путь - это путь вызова, по которому компонент инициатора или межсоединения делает вызовы метода интерфейса вперед в направлении другого компонента межсоединения или цели. Обратный путь - это путь вызова, с помощью которого компонент цели или межсоединения делает обратные вызовы метода интерфейса в направлении другого компонента межсоединения или инициатора. Весь путь между инициатором и мишенью состоит из нескольких переходов, каждый из которых соединяет два соседних компонента. Количество переходов от инициатора к цели больше, чем количество компонентов межсоединений на этом пути. При использовании общей полезной нагрузки путь вперед и назад всегда должен проходить через тот же набор компонентов и сокетов, очевидно, в обратном порядке.

Чтобы поддерживать как прямой, так и обратный пути, для каждого соединения между компонентами требуется порт и экспорт, оба из которых должны быть связаны. Этому способствует сокет инициатора и целевой сокет. Инициатор-сокет предоставляет порт для вызовов метода интерфейса для прямого пути и экспорт для вызовов метода интерфейса на обратном пути. Целевой разъем обеспечивает обратное. Более конкретно, сокет инициатора выводится из класса sc_port и имеет sc_export и наоборот для целевого сокета. Инициатор и классы сокетов-мишеней перегружают оператор привязки порта SystemC, чтобы неявно связывать как прямые, так и обратные пути. Как и транспортные интерфейсы, сокеты также инкапсулируют транспортные интерфейсы DMI и отладки (см. ниже). При использовании сокетов компонент-инициатор будет иметь по меньшей мере один сокет-инициатор, а целевой компонент, по меньшей мере один целевой сокет, и компонент межсоединения по меньшей мере один из них. Компонент может иметь несколько сокетов, переносящих разные типы

транзакций, и в этом случае один компонент может выступать в качестве инициатора или цели для нескольких независимых транзакций.

Для моделирования шинного моста есть две альтернативы. Либо смоделируйте мост шины как компонент межсоединения, либо смоделируйте мост шины как мост транзакции между двумя отдельными транзакциями TLM-2.0. Компонент межсоединения передавал бы указатель на один объект транзакции, что является наилучшим подходом для скорости моделирования. Для моста транзакции требуется копирование объекта транзакции, что дает гораздо большую гибкость, поскольку две транзакции могут иметь разные атрибуты. Использование сокетов TLM-2.0 рекомендуется для максимальной совместимости, удобства и последовательного стиля кодирования. Несмотря на то, что компоненты могут использовать порты SystemC и экспортировать напрямую с основными интерфейсами TLM-2.0, это не рекомендуется.

5.12. Интерфейсы DMI и отладки

Интерфейс прямой памяти (DMI) и интерфейс отладки - это специализированные интерфейсы, отличные от транспортного интерфейса, обеспечивающие прямой доступ и отладочный доступ к области памяти, принадлежащей цели. После предоставления запроса на DMI интерфейс DMI позволяет инициатору обходить обычный путь через компоненты межсоединений, используемые транспортным интерфейсом. DMI предназначен для ускорения транзакций с регулярной памятью в синхронизированной симуляции, тогда как интерфейс отладки транспорта предназначен для отладочного доступа без задержек или побочных эффектов, связанных с регулярными транзакциями. DMI имеет интерфейсы прямой (инициатор к цели) и обратный (цель к инициатору), тогда как отладка имеет только прямой интерфейс.

5.13. Комбинированные интерфейсы и сокеты

Блокирующие и неблокирующие транспортные интерфейсы объединяются с интерфейсами DMI и отладки транспорта в стандартном инициаторе и целевых сокетах. Все четыре интерфейса (два транспортных интерфейса, DMI и debug) могут использоваться параллельно для доступа к заданной цели (в соответствии с правилами, описанными в этом стандарте). Эти объединенные интерфейсы являются одним из ключей к обеспечению взаимодействия между компонентами с использованием стандарта TLM-2.0, а другой ключ - общая полезная нагрузка.

Стандартные целевые сокеты обеспечивают все четыре интерфейса, поэтому каждый целевой компонент должен эффективно реализовывать методы всех четырех интерфейсов. Тем не менее, конструкция блокирую-

щих и неблокирующих транспортных интерфейсов вместе с предоставлением удобных сокетов для преобразования между двумя средствами, которые необходимы для данной цели, только реализуют либо блокирующий, либо неблокирующий транспортный метод, а не оба, в соответствии с требованиями скорости и точности модели. Данный инициатор может выбрать способ вызова через любой или все основные интерфейсы, опять же в соответствии с требованиями скорости и точности. Стили кодирования, упомянутые выше, помогают выбрать подходящий набор функций интерфейса. Как правило, свободно-временной инициатор будет вызывать блокировку транспорта, DMI и отладки, тогда как инициатор с минимальным временем ожидания вызовет неблокирующий транспорт и отладку.

5.14. Пространства имен

Классы TLM-2.0 должны быть объявлены в двух верхних пространствах имен C++, **tlm и tlm_utils**. Частные реализации классов TLM-2.0 могут определять, как будут размещаться другие пространства в этих двух пространствах имен, но такие вложенные пространства имен не должны использоваться в приложениях.

Пространство имен **tlm** содержит классы, которые имеют интерфейс взаимодействия для моделирования шины с отображением памяти. Пространство имен **tlm_utils** содержит классы утилит, которые не являются строго необходимыми для взаимодействия на интерфейсе между моделями шины с памятью, но которые, тем не менее, являются надлежащей частью стандарта TLM-2.0.

5.15. Заголовочные файлы и номера версий

Приложения должны включать (#) заголовочный файл **tlm.h** из каталога **include / tlm** в дистрибутив программного обеспечения. Приложения также должны включать любые файлы заголовков, которые могут потребоваться из каталога **include/tlm/tlm_utils.**

Приложения, составляющие простые сокеты с текущими выпущенными версиями симулятора основной концепции OSCI, должны определить макрос SC_INCLUDE_DYNAMIC_PROCESSES перед включением файла заголовка SystemC.

5.16. Информация о версии программного обеспечения

Файл заголовка включает **/ tlm / tlm_h / tlm_version.h**

И должен содержать набор макросов, констант и функций, которые предоставляют информацию о номере версии программного обеспечения OSCI TLM-2.0. Приложения могут использовать эти макросы и константы.

5.17. Примеры использования основных интерфейсов TLM-2.0

В дополнение к основным интерфейсам TLM-1, TLM-2.0 добавляет блокирующие и неблокирующие транспортные интерфейсы, интерфейс прямой памяти (DMI) и интерфейс отладки транспорта.

5.17.1. Транспортные интерфейсы

Транспортные интерфейсы являются первичными интерфейсами, используемыми для транспортировки транзакций между инициаторами, целями и компонентами межсоединений. Как блокирующие, так и неблокирующие транспортные интерфейсы поддерживают аннотацию времени и временную развязку, но только неблокирующий транспорт поддерживает несколько фаз в течение срока действия транзакции. Блокирующий транспорт не имеет явного аргумента фазы, и любая связь между блокирующим транспортом и фазами неблокирующего транспортного интерфейса является чисто условной. Только неблокирующий метод транспорта возвращает значение, указывающее, был ли использован обратный путь.

Транспортные интерфейсы и общая полезная нагрузка были разработаны для совместного использования в задачах быстрого абстрактного моделирования распределенных по памяти шин. Шаблоны транспортного интерфейса специализируются на типе транзакции, позволяющем использовать их отдельно от общей полезной нагрузки, хотя многие из преимуществ функциональной совместимости будут потеряны. Правила управления памятью объекта транзакции, упорядочения транзакций и допустимой последовательности вызова функций зависят от конкретного типа транзакции, переданного в качестве аргумента шаблона в транспортный интерфейс, который, в свою очередь, зависит от класса признаков протокола, переданного в качестве аргумента шаблона для сокета (если используется сокет).

5.17.2. Блокирующий транспортный интерфейс

Транспортный интерфейс блокировки TLM-2.0 предназначен для поддержки свободно-временного стиля кодирования. Блокирующий транспортный интерфейс является подходящим, когда инициатор хочет завершить транзакцию с целью в течение одного вызова функции и единственными моментами, представляющими интерес, являются те, которые отмечают начало и конец транзакции. Транспортный интерфейс блокировки использует только прямой путь от инициатора к цели.

Транспортный интерфейс блокировки TLM-2.0 использует новый метод `b_transport`, который имеет один аргумент транзакции, переданный неконстантной ссылкой, и второй аргумент для аннотирования времени. Этот единственный аргумент используется как для вызова, так и для

возврата из b_transport, чтобы указать время начала и конца транзакции, соответственно, относительно текущего времени моделирования.

5.17.3. Определение класса блокирующего интерфейса

Для определения класса блокирующего интерфейса используют следующий синтаксис:

Листинг 5.11

```
    namespace tlm { template <typename TRANS =
tlm_generic_payload>
    class tlm_blocking_transport_if : public virtual
sc_core::sc_interface { public:
    virtual void b_transport(TRANS& trans,
sc_core::sc_time& t) = 0;
    };
    } // namespace tlm
```

5.17.4. Аргумент шаблона TRANS

Этот основной интерфейс может использоваться для транспортировки транзакций любого типа. Конкретный тип транзакции tlm_generic_payload предоставляется для облегчения взаимодействия между моделями, где точные детали атрибутов транзакций менее важны.

Для максимальной совместимости приложения должны использовать стандартный тип транзакции tlm_generic_payload с базовым протоколом. Чтобы моделировать конкретные протоколы, приложения могут заменять свой собственный тип транзакции. Сокеты, которые используют интерфейсы, специализированные для разных типов транзакций, не могут быть связаны друг с другом. Они обеспечивают проверку времени компиляции, но ограничивают совместимость.

5.17.5. Правила

А) Метод b_transport может вызвать прямое или косвенное оповещение.

Б) Метод b_transport не вызывается из процесса метода.

В) Инициатор может повторно использовать объект транзакции от одного вызова до следующего и через вызовы к транспортным интерфейсам, DMI и транспортному интерфейсу отладки.

Г) Вызов b_transport отмечает первую точку синхронизации транзакции. Возврат из b_transport указывает конечную точку синхронизации транзакции.

Д) Аргумент аннотации времени позволяет сместить точки синхронизации с времен моделирования (значение, возвращаемое `sc_time_stamp ()`), с которого выполняются вызов функции и возврат.

Е) Вызываемый может изменять или обновлять объект транзакции с учетом любых ограничений, налагаемых транзакционным классом TRANS.

Ж) Рекомендуется, чтобы объект транзакции не содержал информацию о времени. Сроки должны быть аннотированы с использованием аргумента `sc_time` для `b_transport`.

З) Как правило, компонент межсоединения должен передавать вызов `b_transport` по прямому пути от инициатора к цели. Другими словами, реализация `b_transport` для целевого сокета компонента межсоединения может вызывать метод `b_transport` для сокета-инициатора.

И) Независимо от того, разрешено ли выполнение `b_transport`, вызов `nb_transport_fw`, зависит от правил, связанных с протоколом. Для базового протокола удобный сокет `simple_target_socket` может автоматически выполнять это преобразование.

К) Реализация `b_transport` не должна вызывать `nb_transport_bw`.

5.17.6. График последовательности сообщений - блокировка транспорта

Метод блокирующего транспорта может немедленно возвращаться (а именно, на текущую фазу оценки SystemC) или может дать управление планировщику и вернуться к инициатору только в более позднюю точку во время моделирования. Хотя поток инициатора может быть заблокирован, другому потоку в инициаторе может быть разрешено вызывать `b_transport` до того, как первый вызов вернется, в зависимости от протокола.

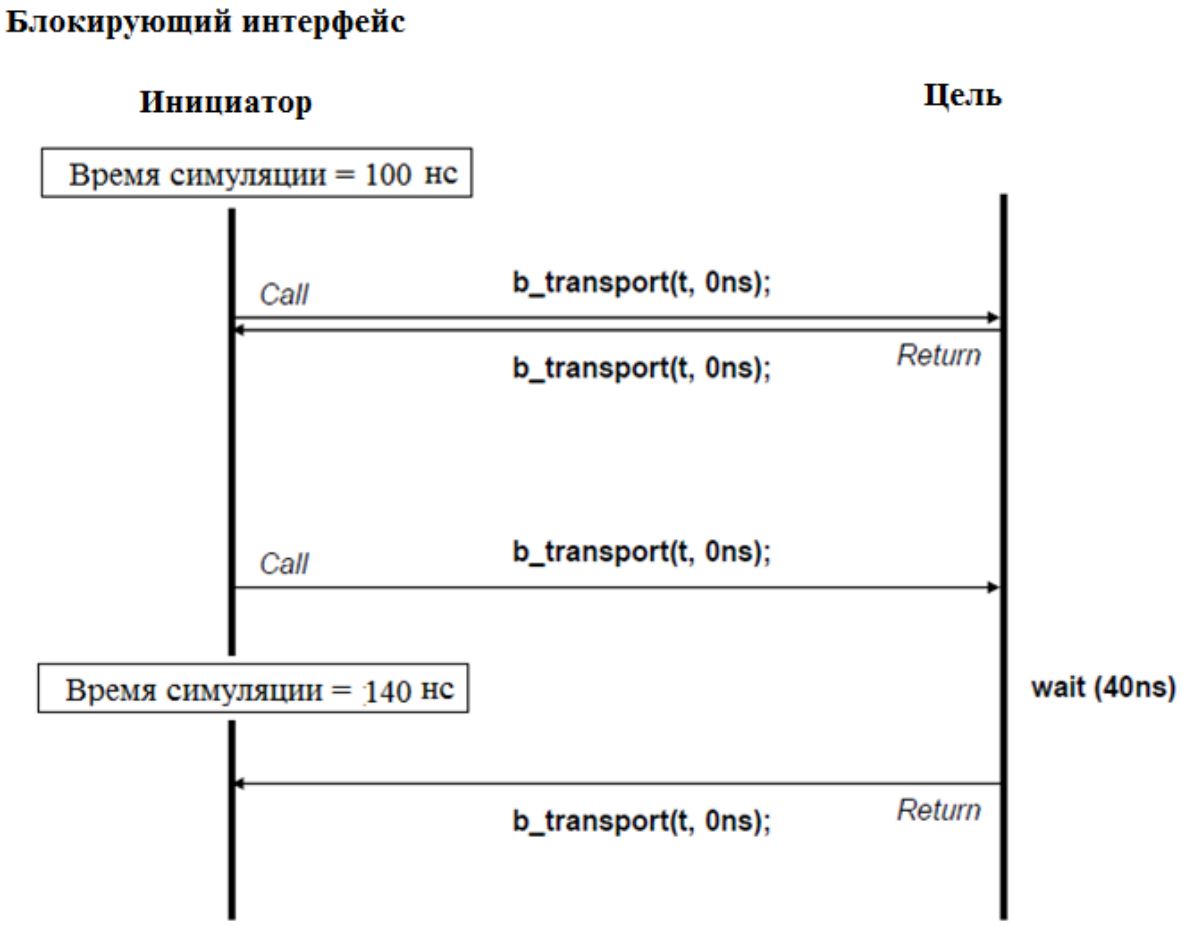


Рис. 5.5. Диаграмма работы блокирующего интерфейса

5.17.6. График последовательности сообщений - временная развязка

Временно развязанный инициатор может работать в условное локальное время перед текущим временем моделирования, и в этом случае он должен передать ненулевое значение для аргумента времени для `b_transport`, как показано ниже. Каждая транзакция может иметь свою длительность выполнения. Инициатор и цель могут каждый дополнительно продвигать локальное смещение по времени, увеличивая значение аргумента времени. Добавление аргумента времени, возвращаемого вызовом на текущее время моделирования, дает условное время, в которое каждая транзакция завершается, но само время моделирования не может продвигаться до тех пор, пока не будет получен поток инициатора. Тело `b_transport` может само вызвать `wait`, и в этом случае местное смещение времени должно быть сброшено до нуля. На приведенной ниже диаграмме окончательный возврат от инициатора происходит во время моделирования 140ns, но с аннотированной задержкой в 5 нс, давая эффективное местное время 145 нс.

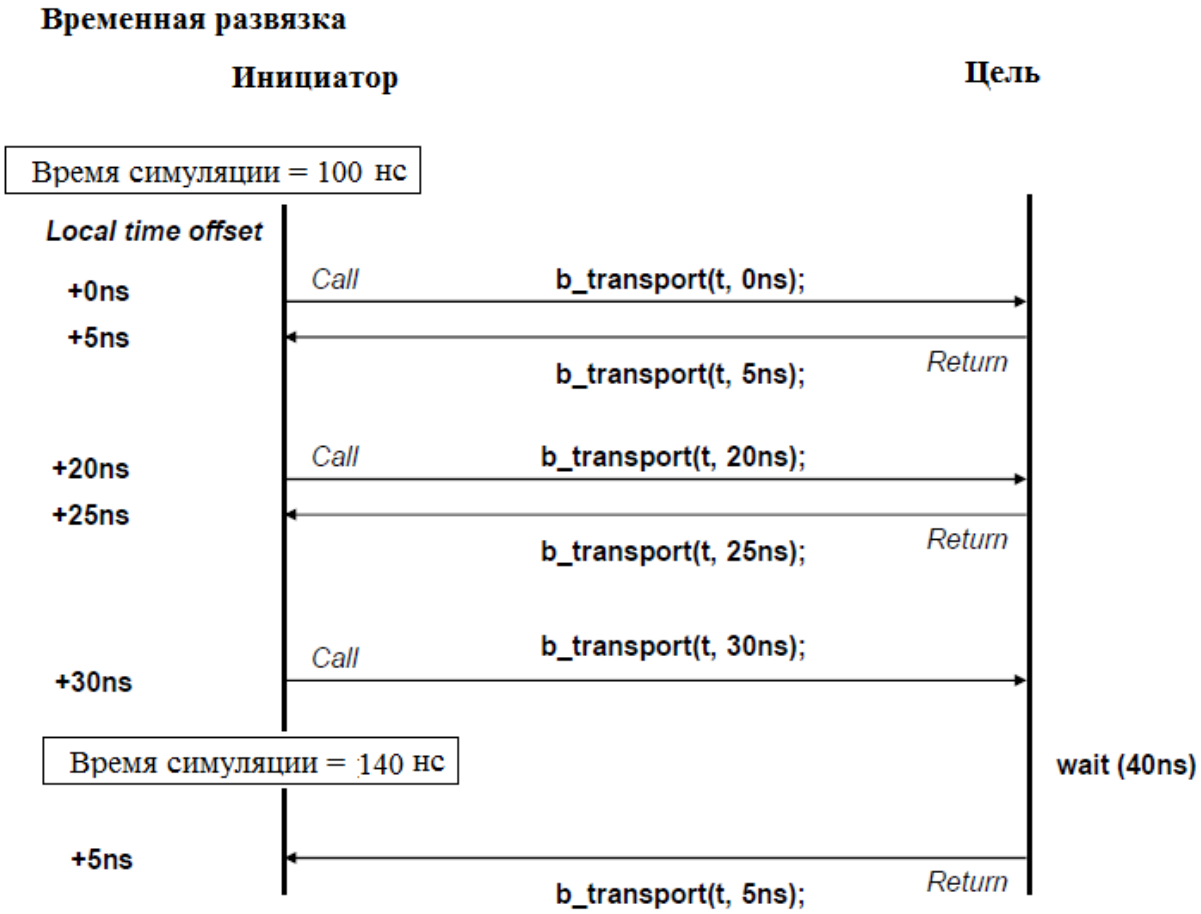


Рис. 5.6. Диаграмма интерфейса с временной развязкой

5.17.7. Схема последовательности сообщений – квантование времени

Временно - развязанный инициатор будет продолжать продвигать локальное время до тех пор, пока квант времени не будет превышен. В этот момент инициатор обязан синхронизировать, приостанавливая выполнение, прямо или косвенно вызывая метод `wait` с локальным временем в качестве аргумента. Это позволяет другим инициаторам в модели запускать и получать ответы, что эффективно означает, что инициаторы выполняются поочередно, каждый из которых несет ответственность за определение того, когда нужно отменить контроль, отслеживая свое местное время. Первоначальный инициатор должен запускаться только после того, как время моделирования перешло к следующему кванту. Основная цель задержек в стиле свободно-временного кодирования - позволить каждому инициатору определить, когда нужно отменить управление. Для правильной работы лучше, если модель не будет полагаться на детали времени. Внутри каждого кванта транзакции, генерируемые данным инициатором, происходят в строгом последовательном порядке, но без увеличения вре-

мени моделирования. Местное время не отслеживается планировщиком SystemC.

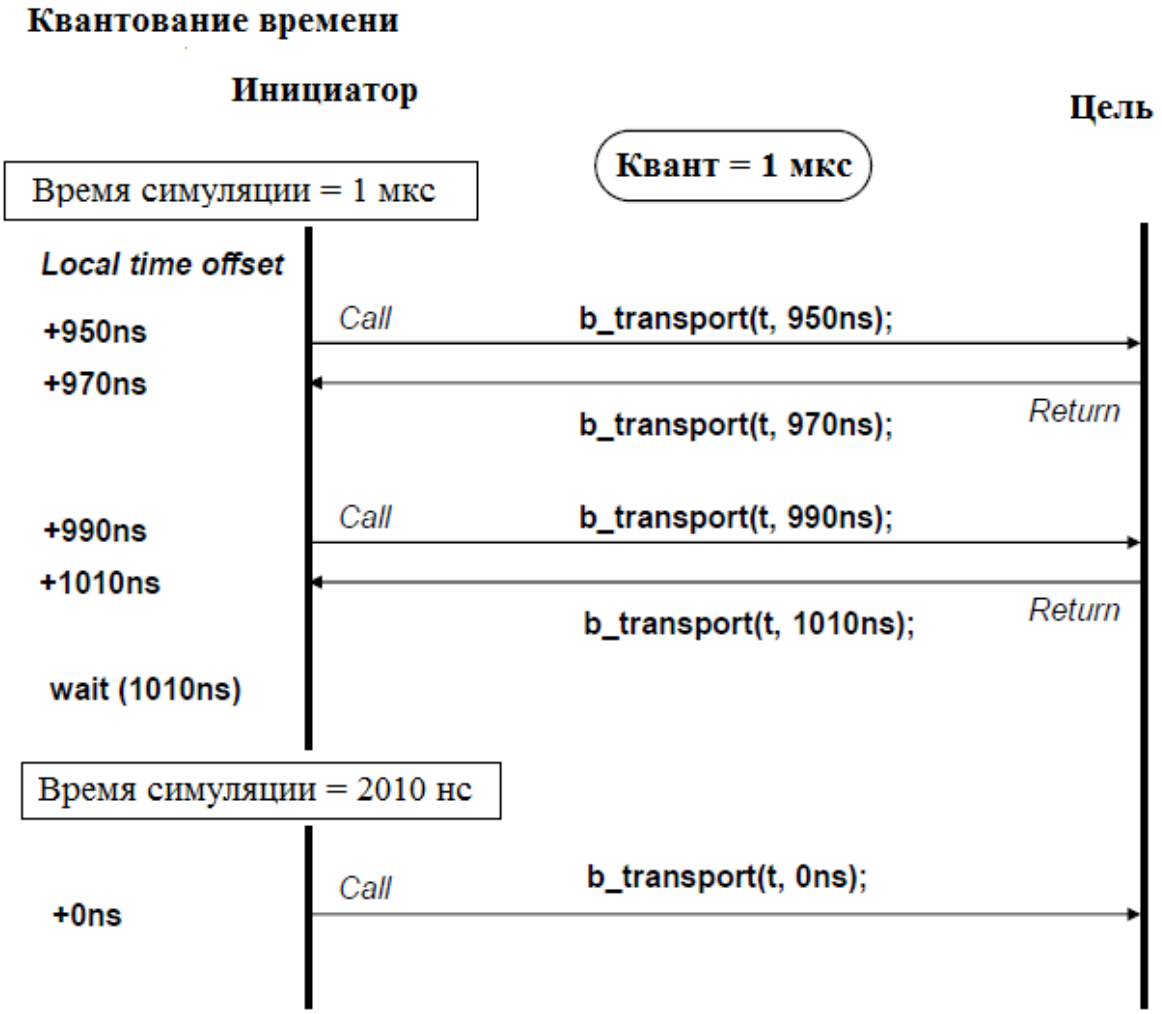


Рис. 5.7. Диаграмма последовательности транзакций с квантованным временем

5.17.8. Неблокирующий транспортный интерфейс

Неблокирующий транспортный интерфейс предназначен для поддержки приближенного по времени стиля кодирования. Неблокирующий транспортный интерфейс является подходящим, когда желательно моделировать подробную последовательность взаимодействий между инициатором и целью в течение каждой транзакции. Другими словами, чтобы разбить транзакцию на несколько фаз, где каждый фазовый переход связан с точкой синхронизации. Каждый вызов и возврат из неблокирующего транспортного метода может соответствовать фазовому переходу. Ограничивая количество точек синхронизации до двух, можно использовать не-

блокирующий транспортный интерфейс со свободно-временным стилем кодирования, но это обычно не рекомендуется. Для свободно-временного моделирования блокирующий транспортный интерфейс обычно предпочтительнее из-за его простоты. Неблокирующий транспортный интерфейс особенно подходит для моделирования конвейерных транзакций, что было бы неудобно моделировать, используя блокирующий транспорт.

Неблокирующий транспортный интерфейс использует как прямой путь от инициатора к цели, так и как обратный путь от цели до инициатора. Существует два разных интерфейса: `tlm_fw_nonblocking_transport_if` и `tlm_bw_nonblocking_transport_if` для использования на противоположных путях. Неблокирующий транспортный интерфейс использует аналогичный механизм передачи аргументов как в транспортном интерфейсе блокировки тем, что неблокирующие транспортные методы передают неконстантную ссылку на объект транзакции и аннотацию времени, но на этом заканчивается сходство. Неблокирующий метод транспорта также передает фазу, указывающую состояние транзакции, и возвращает значение перечисления, чтобы указать, является ли возврат из функции также фазовым переходом. Как блокировка, так и неблокирующая передача поддерживает аннотацию времени, но только неблокирующий транспорт поддерживает несколько фаз в течение жизни транзакции. Блокирующий и неблокирующий транспортный интерфейс и общая полезная нагрузка были разработаны для совместного использования для задач быстрого абстрактного моделирования подключенных к памяти шин. Однако, транспортные интерфейсы могут использоваться отдельно от общей полезной нагрузки для конкретных моделей протоколов. Тип транзакции и тип фазы являются параметрами шаблона неблокирующего транспортного интерфейса

5.17.9. Определение класса неблокирующего интерфейса

Для определения класса неблокирующего интерфейса используют следующий синтаксис:

Листинг 5.12

```
    namespace tlm {
    enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED,
TLM_COMPLETED };
    template <typename TRANS = tlm_generic_payload,
typename PHASE = tlm_phase>
    class tlm_fw_nonblocking_transport_if : public
virtual sc_core::sc_interface {
    public:
```

```
    virtual tlm_sync_enum nb_transport_fw(TRANS&
trans, PHASE& phase, sc_core::sc_time& t) = 0;
     };
    template <typename TRANS = tlm_generic_payload,
typename PHASE = tlm_phase>
    class tlm_bw_nonblocking_transport_if : public
virtual sc_core::sc_interface {
    public:
    virtual tlm_sync_enum nb_transport_bw(TRANS&
trans, PHASE& phase, sc_core::sc_time& t) = 0;

    };
    } // namespace tlm
```

5.17.10. Аргументы шаблона TRANS и PHASE

Неблокирующий транспортный интерфейс может использоваться для транспортировки транзакций любого типа и с любым количеством фаз и временных точек. Определенный тип транзакции `tlm_generic_payload` предоставляется для облегчения взаимодействия между моделями, где точные данные атрибутов транзакции менее важны, и для использования с базовым протоколом предоставляется определенный тип `tlm_phase`.

Для максимальной совместимости приложения должны использовать тип транзакции по умолчанию `tlm_generic_payload` и тип фазы `tlm_phase` по умолчанию с базовым протоколом. Чтобы моделировать конкретные протоколы, приложения могут заменять свой тип транзакции и тип фазы. Сокеты, которые используют интерфейсы, специализированные для разных типов транзакций, не могут быть связаны друг с другом, обеспечивая проверку времени компиляции, но ограничивая совместимость.

5.17.11. Запросы nb_transport_fw и nb_transport_bw

А) Существует два неблокирующих метода транспорта: **nb_transport_fw** для использования по прямому пути и **nb_transport_bw** для использования на обратном пути. Помимо их названий и направления вызова эти два метода имеют сходную семантику. Ниже курсивный термин *nb_transport* используется для описания обоих методов в ситуациях, когда нет необходимости различать их.

Б) В случае базового протокола, прямой и обратный пути должны проходить через точно такую же последовательность компонентов и сокетов в противоположном порядке. Каждый компонент несет ответственность за маршрутизацию любой транзакции, возвращающейся к инициато-

ру, с использованием целевого сокета, через который эта транзакция была впервые получена.

В) **nb_transport_fw** вызывается только по прямому пути, а **nb_transport_bw** должен вызываться только на обратном пути.

Г) Вызов **nb_transport_fw** по прямому пути ни при каких обстоятельствах не должен прямо или косвенно обращаться к **nb_transport_bw** по обратному пути и наоборот.

Д) Методы *nb_transport* не должны вызывать `wait`, прямо или косвенно.

Е) Методы *nb_transport* могут быть вызваны из процесса потока или из процесса метода.

Ж) *nb_transport* не разрешено вызывать **b_transport.** Одним из решений было бы вызвать **b_transport** из отдельного процесса потока, порожденного или уведомленного исходным методом **nb_transport_fw**. Для базового протокола предоставляется удобный сокет **simple_target_socket**, который может автоматически выполнять это преобразование.

З) Неблокирующий транспортный интерфейс явно предназначен для поддержки конвейерных транзакций. Другими словами, несколько последовательных вызовов **nb_transport_fw** из одного процесса могут инициировать отдельные транзакции, не дожидаясь завершения первой транзакции.

И) В принципе, конечная точка синхронизации транзакции может быть отмечена вызовом или возвратом из *nb_transport* либо по прямому пути, либо по обратному пути.

5.17.12. Аргумент trans

А) Время жизни данного объекта транзакции может выходить за пределы возврата из *nb_transport*, так что серия вызовов *nb_transport* может передавать один объект транзакции вперед и назад между инициаторами, компонентами соединения и целями.

Б) Если есть несколько вызовов *nb_transport*, связанных с данным экземпляром транзакции, один и тот же объект транзакции передается в качестве аргумента для каждого такого вызова. Другими словами, данный экземпляр транзакции должен быть представлен одним объектом транзакции.

В) Инициатор может повторно использовать данный объект транзакции для представления более одного экземпляра транзакции или вызовов к транспортным интерфейсам, интерфейсу DMI и интерфейсу отладки.

Г) Поскольку время жизни объекта транзакции может распространяться на несколько вызовов *nb_transport*, либо вызывающий, либо вызываемый могут изменять или обновлять объект транзакции с учетом любых ограничений, налагаемых транзакционным классом TRANS. Например,

для общей полезной нагрузки целевой объект может обновить массив данных объекта транзакции в случае команды чтения, но не должен обновлять поле команды.

5.17.13. Фазовый аргумент

А) Каждый вызов *nb_transport* передает ссылку на **фазовый объект**. В случае базового протокола не разрешаются последовательные вызовы *nb_transport* с той же фазой. Каждый фазовый переход имеет связанную временную точку. Аннотации времени с использованием аргумента `sc_time` должны задерживать точку синхронизации относительно фазового перехода.

Б) Фазовый аргумент передается по ссылке. Либо вызывающий, либо вызываемый могут изменять фазу.

В) Цель состоит в том, чтобы аргумент фазы использовался для информирования компонентов о том, разрешено ли и когда разрешено читать или изменять атрибуты транзакции. Если правила протокола позволяют данному компоненту изменять значение атрибута транзакции в течение конкретной фазы, то этот компонент может изменять значение в любое время на этой фазе и любое количество раз в течение этой фазы. Протокол должен запрещать другим компонентам считывать значение этого атрибута на этой фазе, только разрешая считывание значения после следующего фазового перехода.

Г) Значение аргумента фазы представляет текущее состояние конечного автомата протокола для данного «скачка». Если один объект транзакции передается между более чем двумя компонентами (инициатором, межсоединением, целевым), каждый «скачок» требует (как минимум, по меньшей мере) отдельного конечного автомата протокола.

Д) Если объект транзакции имеет продолжительность жизни и область действия, которая может простираться за пределы одного вызова *nb_transport*, фазовый объект обычно является локальным для вызывающего. Каждый вызов *nb_transport* для данной транзакции может иметь отдельный фазовый объект. Соответствующие фазовые переходы на разных скачках могут возникать в разных точках времени моделирования.

Е) Тип фазы по умолчанию `tlm_phase` специфичен для базового протокола. Другие протоколы могут использовать или расширять тип `tlm_phase` или могут заменить их на собственный тип фазы (с соответствующей потерей функциональной совместимости).

Класс `tlm_phase` - это тип фазы по умолчанию, используемый шаблонами класса неблокирующего транспортного интерфейса и базовым протоколом. Объект `tlm_phase` представляет фазу с неподписанным значением `int`. Класс `tlm_phase` - это присвоение, совместимое с ти-

пом `unsigned int` и с перечислением, имеющим значения, соответствующие четырем фазам базового протокола, а именно BEGIN_REQ, END_REQ, BEGIN_RESP и END_RESP:

- **BEGIN_REQ** (начало запроса)
  - Инициатор приобретает шину
  - Соединения становятся «занятыми» и блокируют дальнейшие запросы
  - Полезная нагрузка становится «занятой»
- **END_REQ** (конец запроса)
  - Цель «принимает» запрос и завершает «рукопожатие»
  - Шина освобождена для запуска дополнительных запросов
- **BEGIN_RESP** (начало ответа)
  - Цель получает шину для ответа
  - Шина становится «занятой»
- **END_RESP** (конец ответа)
  - Инициатор подтверждает ответ, чтобы завершить его
  - Шина свободна
  - Ссылка на полезную нагрузку освобождается

Поскольку тип `tlm_phase` является классом, а не перечислением, он может поддерживать перегруженный оператор потока, чтобы отображать значение фазы как текст ASCII.

Набор из четырех фаз, предоставляемых `tlm_phase_enum`, может быть расширен с помощью макроса DECLARE_EXTENDED_PHASE. Этот макрос создает одноэлементный класс, полученный из `tlm_phase`, с помощью метода `get_phase`, который возвращает соответствующий объект. Этот объект может использоваться как новый этап. Для максимальной совместимости приложение должно использовать только четыре фазы `tlm_phase_enum`. Если для моделирования деталей конкретного протокола требуются дополнительные фазы, предполагается, что следует использовать DECLARE_EXTENDED_PHASE, поскольку это сохраняет совместимость присваивания с типом `tlm_phase`.

Принцип *неосведомляемого* и необратимого или обязательного расширения применяется к фазам так же, как к общим расширениям полезной нагрузки. Другими словами, *неосведомленные* фазы разрешены базовым протоколом. Получатель должен игнорировать неосведомленную фазу в том смысле, что получатель может просто действовать так, как если бы он не получил фазовый переход, и, следовательно, отправитель должен был иметь возможность продолжать в отсутствие ответа от получателя. Если в этом смысле фаза не игнорируется, должен быть определен новый класс признаков протокола.

5.17.14. Возвращаемое значение tlm_sync_enum

А) Понятие сихронизации упоминается в нескольких местах. Синхронизация заключается в том, чтобы дать управление планировщику SystemC и чтобы другие процессы могли выполняться, но имеет дополнительные значения для временной развязки.

Б) В принципе, синхронизация может быть выполнена путем уступки (вызов ожидания в случае процесса потока или возврат к ядру в случае процесса метода), но временный отключаемый инициатор должен синхронизировать, вызывая метод синхронизации класса `tlm_quantum_keeper` . В общем случае инициатору необходимо время от времени синхронизировать, чтобы разрешить выполнение других процессов SystemC.

В) Следующие правила применяются как для прямого, так и для обратного пути.

Г) Значение возвращаемого значения из `nb_transport` фиксировано и не изменяется в зависимости от типа транзакции или типа фазы. Следовательно, следующие правила не ограничиваются базовым протоколом, а применяются к каждой транзакции и типу фазы, используемому для параметризации неблокирующего транспортного интерфейса.

Д) TLM_ACCEPTED. Вызывающая сторона не должна изменять состояние объекта транзакции, фазы или аргумента времени во время вызова. Другими словами, TLM_ACCEPTED указывает, что путь возврата не используется. Вызывающий может игнорировать значения аргументов *nb_transport* после вызова, так как вызываемый обязан оставить их неизменными. В общем случае вызывающий абонент должен будет уступить, прежде чем компонент, содержащий вызываемого абонента, сможет ответить на транзакцию. Для базового протокола вызывающий, игнорирующий фазу, должен возвращать TLM_ACCEPTED.

Е) TLM_UPDATED. Вызов обновил объект транзакции. Вызывающий может изменить состояние аргумента фазы, возможно, изменил состояние объекта транзакции и, возможно, увеличил значение аргумента времени во время вызова. Другими словами, TLM_UPDATED указывает, что используется обратный путь, и вызываемый пользователь расширил состояние конечного автомата протокола, связанного с транзакцией, независимо от того, действительно ли вызывающая сторона обязана изменять каждый из аргументов, что зависит от протокола. После вызова *nb_transport* вызывающий должен проверить аргументы фазы, транзакции и времени и предпринять соответствующие действия.

Ж) TLM_COMPLETED. Вызов обновил объект транзакции, и транзакция завершена. Вызывающий может изменить состояние объекта транзакции и, возможно, увеличил значение аргумента времени во время вызо-

ва. Значение аргумента фазы не определено. Другими словами, TLM_COMPLETED указывает, что используется обратный путь, и транзакция завершена в отношении конкретного сокета. После вызова *nb_transport* вызывающий должен проверить объект транзакции и предпринять соответствующие действия, но должен игнорировать фазовый аргумент. Не должно быть никаких транспортных вызовов, связанных с этой конкретной транзакцией, через текущий сокет вдоль прямого или обратного пути. Завершение в этом смысле необязательно означает успешное завершение, поэтому в зависимости от типа транзакции вызывающему может потребоваться проверить статус ответа, встроенный в объект транзакции.

З) В общем случае нет обязательств по завершению транзакции путем возврата *nb_transport* TLM_COMPLETED. В любом случае транзакция завершена в отношении конкретного сокета, когда последняя фаза протокола передается в качестве аргумента для *nb_transport*. (Для базового протокола конечная фаза - END_RESP.) Другими словами, TLM_COMPLETED не является обязательным.

И) Для любого из трех возвращаемых значений и в зависимости от протокола после вызова *nb_transport* вызывающему абоненту, возможно, придется уступить, чтобы позволить компоненту, содержащему вызываемого абонента, генерировать ответ или выпустить объект транзакции.

tlm_sync_enum	**Объект транзакции**	**Фаза по возвращении**	**Временная аннотация по возвращении**
TLM_ACCEPTED	Неизменный	Без изменений	Без изменений
TLM_UPDATE	Обновленный	Измененный	Может быть увеличено
TLM_COMPLETED	Обновленный	Игнорируется	Может быть увеличено

5.17.15. Диаграмма последовательности сообщений – использование пути назад

Следующие диаграммы последовательности сообщений иллюстрируют различные последовательности вызовов для *nb_transport*. Аргументы и возвращаемое значение, переданные в или из *nb_transport*, показаны с использованием возврата описания `return, phase, delay`, где `return` - это значение, возвращаемое из вызова функции, `phase` - это

значение аргумента фазы, а `delay` - значение аргумента `sc_time`. Обозначение ,_, указывает, что значение не используется.

Следующие диаграммы последовательности сообщений используют фазы базового протокола в качестве примера, то есть BEGIN_REQ, END_REQ и т.д . С приближенным по времени стилем кодирования и базовым протоколом транзакция передается обратно и вперед дважды между инициатором и целью. Для других протоколов количество фаз и их имена могут быть разными.

Если получатель вызова *nb_transport* не может сразу вычислить следующее состояние транзакции или задержку до следующей точки синхронизации, он должен вернуть значение TLM_ACCEPTED. Вызывающий абонент должен получить контроль над планировщиком и ожидать получения вызова *nb_transport* на противоположном пути, когда вызываемый пользователь готов ответить. Обратите внимание, что в этом случае, в отличие от случая с ограниченным сроком, вызывающий может быть инициатором или целью.

Транзакции могут быть конвейерными. Инициатор мог вызвать *nb_transport* для отправки другой транзакции цели, прежде чем увидеть последний фазовый переход предыдущей транзакции. Поскольку процессы регулярно дают контроль над планировщиком, чтобы обеспечить время моделирования, приближенный стиль кодирования, как ожидается, будет симулировать намного медленнее, чем стиль с свободно- временным кодирования.

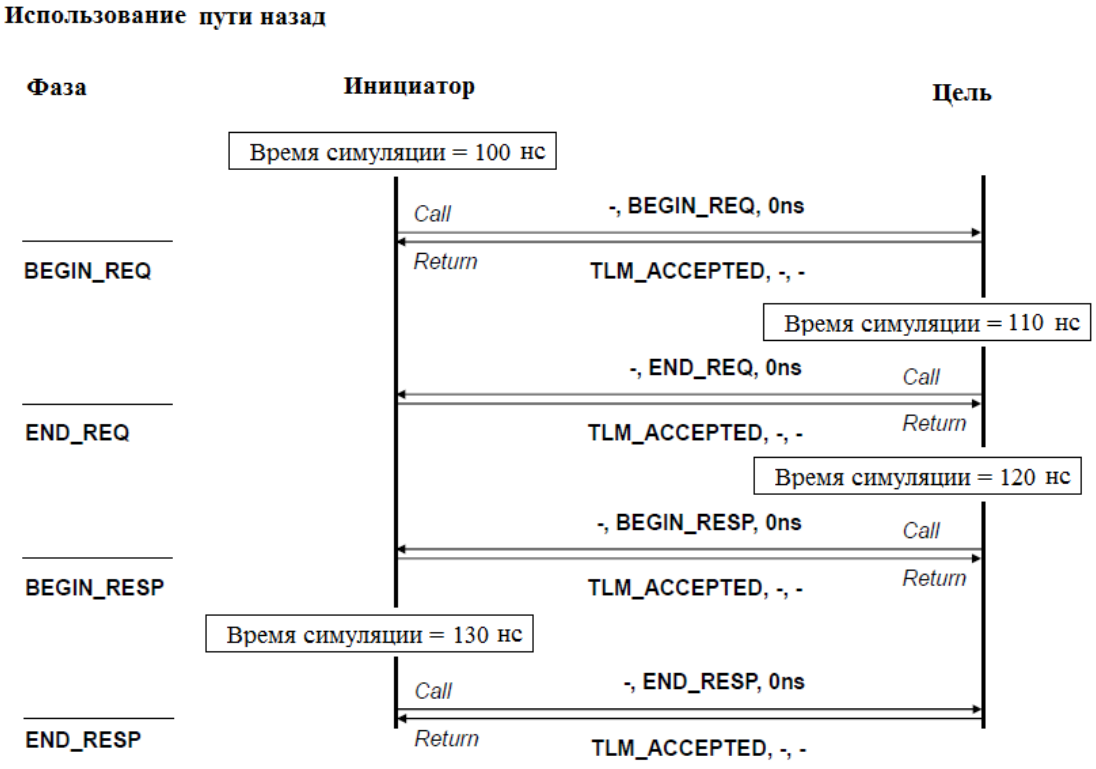


Рис. 5.8. Диаграмма последовательности сообщений с использованием пути назад

5.17.16. Схема последовательности сообщений - с использованием обратного пути

Если получатель вызова *nb_transport* может сразу вычислить следующее состояние транзакции и задержку до следующей точки синхронизации, он может вернуть новое состояние по возврату из *nb_transport*, а не использовать противоположный путь. Если следующая точка синхронизации отмечает конец транзакции, получатель может вернуть либо TLM_UPDATED, либо TLM_COMPLETED. Вызов может возвращать TLM_COMPLETED на любом этапе (в соответствии с правилами протокола), чтобы указать вызывающему, чтобы он предварительно освободил другие этапы и перешел на заключительный этап, завершив транзакцию. Это относится как к инициатору, так и к цели. С TLM_UPDATED вызываемый должен обновить транзакцию, фазу и аннотацию времени. На приведенной ниже диаграмме ненулевой аргумент аннотации времени, переданный при возврате из вызовов функций, указывает вызывающему абоненту задержку между фазовым переходом на скачке и соответствующей точкой синхронизации.

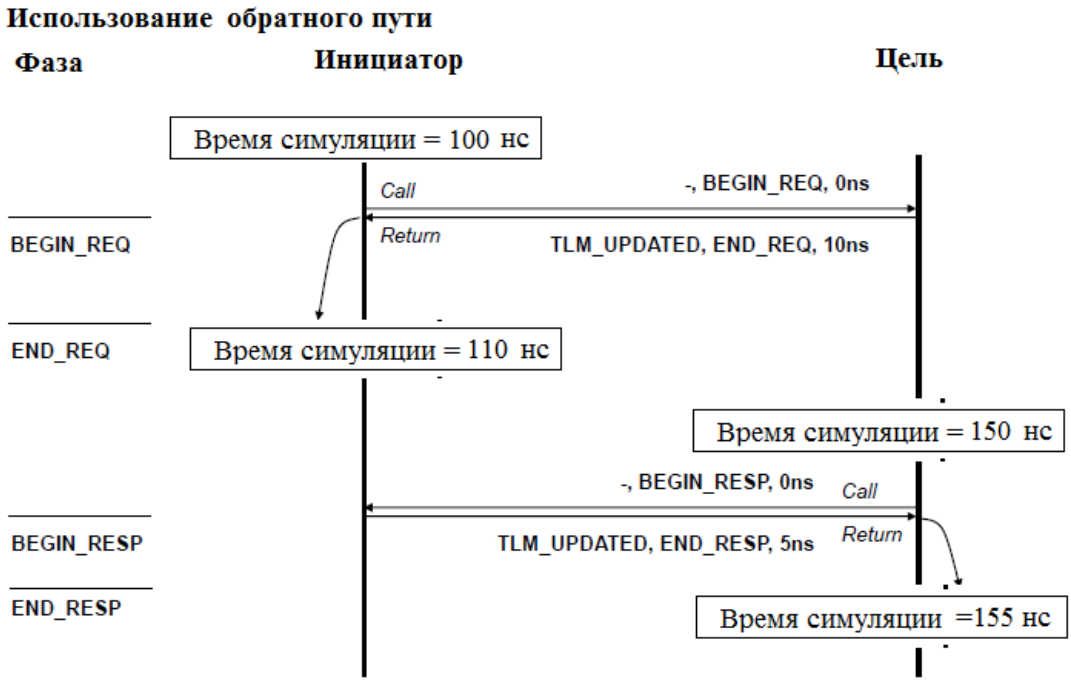


Рис. 5.9. Диаграмма последовательности сообщений с использованием обратного пути

5.17.17. Диаграмма последовательности сообщений – раннее завершение

В зависимости от протокола инициатор или цель могут возвращать TLM_COMPLETED из *nb_transport* в любой момент, чтобы завершить транзакцию раньше. Ни инициатор, ни цель не могут совершать какие-либо дальнейшие вызовы *nb_transport* для этого конкретного экземпляра транзакции. Независимо от того, разрешено ли инициатору или цели сократить транзакцию таким образом, этот путь зависит от правил конкретного протокола. На приведенной ниже диаграмме аннотация времени по пути возврата указывает инициатору, что конечная точка синхронизации должна произойти после заданной задержки. Фазовые переходы из BEGIN_REQ через END_REQ и BEGIN_RESP в END_RESP являются неявными, а не передаются явно как аргументы *nb_transport*.

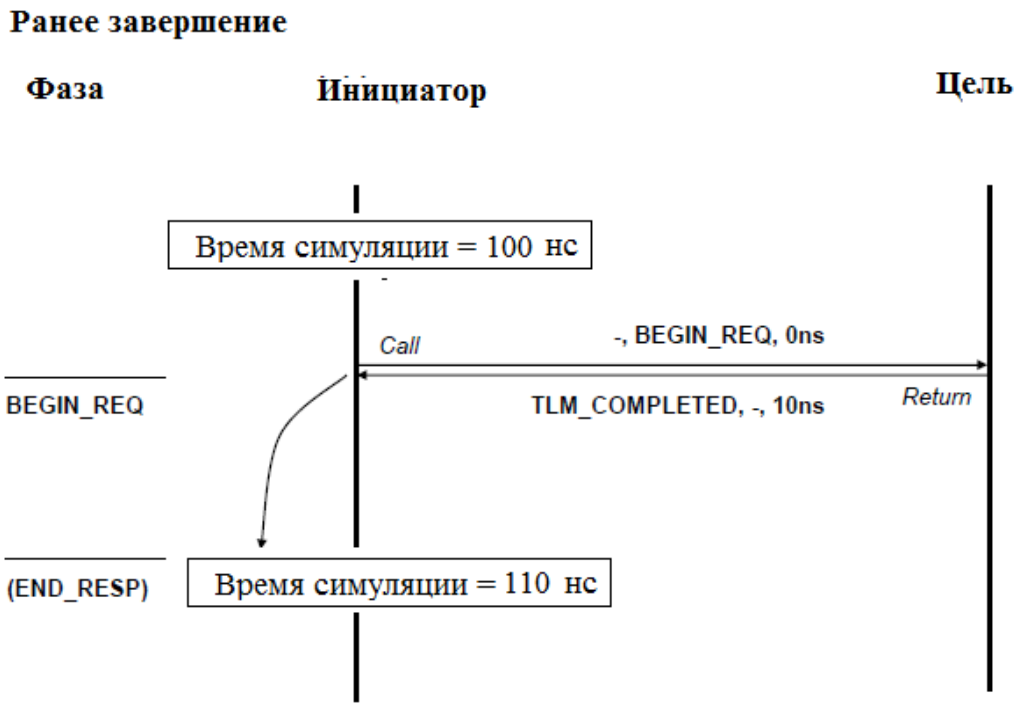


Рис. 5.10. Диаграмма последовательности сообщений для раннего завершение

5.17.18. График последовательности сообщений – аннотация времени

Вызывающий может аннотировать задержку на вызов *nb_transport*. Это является признаком того, что транзакция должна обрабатываться так, как если бы она была получена после указанной задержки. Приближенный по времени расчетный вызов обычно обрабатывает эту ситуацию, помещая

транзакцию в очередь событий полезной нагрузки для обработки, когда время моделирования увязывается с аннотированной задержкой. В зависимости от реализации очереди событий полезной нагрузки эта обработка может происходить либо в процессе SystemC, который чувствителен к уведомлению о событии из очереди событий полезной нагрузки, либо в обратном вызове, зарегистрированном в очереди событий полезной нагрузки. Задержки могут быть аннотированы на вызовы прямого и обратного путей и соответствующие пути возврата. Ожидается, что инициатор с минимальным временем ожидания будет обрабатывать входящие транзакции как для прямого пути возврата, так и для обратного пути таким же образом. Точно так же ожидается, что приближенно-временная задача будет обрабатывать входящие транзакции как по обратному пути возврата, так и по прямому пути таким же образом.

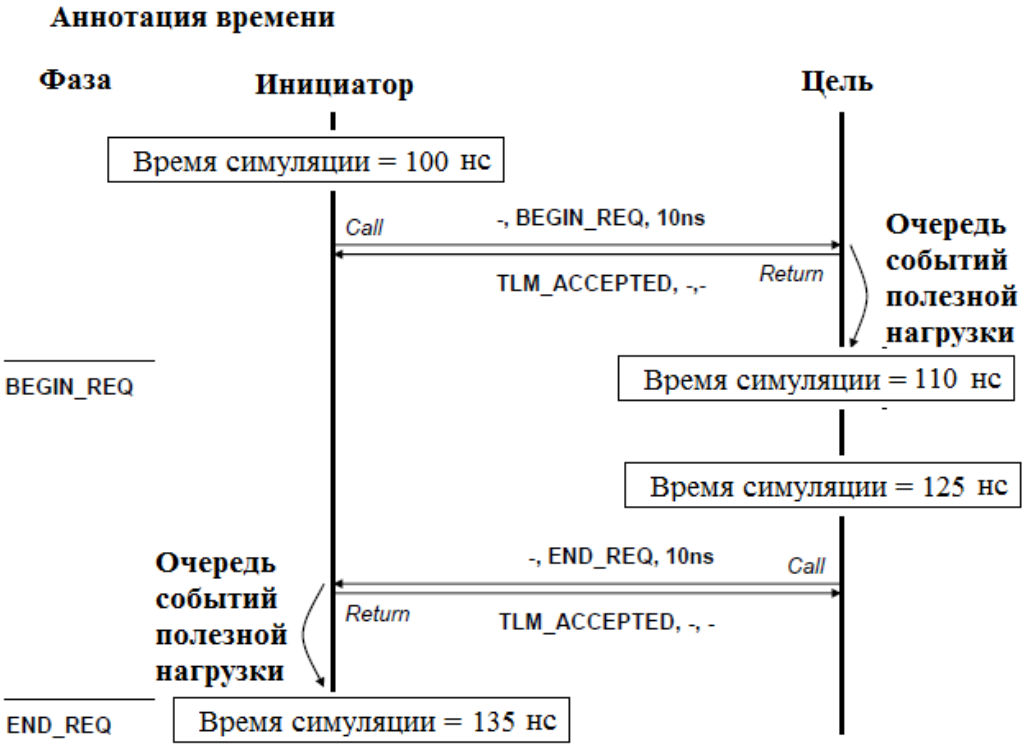


Рис. 5.11. Диаграмма последовательности сообщений с аннотацией времени

5.17.19. Аннотации синхронизации с транспортными интерфейсами

Аннотирование синхронизации - это общая функция блокирующих и неблокирующих транспортных интерфейсов, выраженная с использованием аргумента **sc_time** для методов **b_transport**, **nb_transport_fw** и **nb_transport_bw.** В этом документе курсивный термин *transport* исполь-

зуется для обозначения трех методов **b_transport**, **nb_transport_fw** и **nb_transport_bw.**

Порядок транзакций определяется комбинацией правил интерфейса ядра и правил протокола. Правила в следующем разделе применяются к основным интерфейсам независимо от выбора протокола.

Аргумент `sc_time`

Рекомендуется, чтобы объект транзакции не содержал информацию о времени. Любые аннотации времени должны быть выражены с использованием аргумента `sc_time` для переноса.

Аргумент времени должен быть неотрицательным и должен быть выражен относительно текущего времени моделирования `sc_time_stamp ().`

Аргумент времени применяется как к вызову, так и к возврату из транспорта (в соответствии с правилами возвращаемого значения `tlm_sync_enum nb_transport`).

Метод **nb_transport** может сам увеличить значение аргумента времени, но не должен уменьшать значение. Метод **b_transport** может увеличить значение аргумента `time` или может уменьшить значение в случае, когда он вызвал ожидание и, таким образом, синхронизирован с временем моделирования, но только на сумму, которая меньше или равна времени, для которого процесс был приостановлен. Это правило согласуется со временем, которое не выполняется в обратном направлении в симуляции SystemC.

Документ OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL содержит более 20 правил и инструкций по использованию аргументов времени и аннотаций времени.

5.17.20. Примеры стилей кодирования

Следующие фрагменты псевдокода иллюстрируют только некоторые из многих возможных стилей кодирования:

Листинг 5.13

```
    // Различные определения метода интерфейса
    void b_transport( TRANS& trans, sc_core::sc_time&
t )
    {
     // Свободно-временной  стиль кодирования
execute_transaction( trans );
    t = t + latency;
    }
```

```
    void b_transport( TRANS& trans, sc_core::sc_time&
t )
    {

    //-----------------------
    /*Свободно-временной  с синхронизацией на цели
или ожидании синхронизации по требованию ожидания
wait(t)*/;
    execute_transaction( trans );
    t = SC_ZERO_TIME;
    }

    tlm_sync_enum nb_transport_fw( TRANS& trans,
PHASE& phase, sc_core::sc_time& t )
    {

    //----------------------
    /* Псевдо-некорректный стиль кодирования с
использованием неблокирующего транспорта (не
рекомендуется)*/
     execute_transaction( trans );
    t = t + latency;
     return TLM_COMPLETED;
    }

    tlm_sync_enum nb_transport_fw( TRANS& trans,
PHASE& phase, sc_core::sc_time& t )
    {

    //-----------------------
    // Приближенно-временной стиль кодирования
    /*Отправьте транзакцию в очередь событий полезной
нагрузки для выполнения во время sc_time_stamp () + t
*/
    payload_event_queue->notify( trans, phase, t );
     // Увеличение количества ссылок на транзакцию
     trans.acquire();
    return TLM_ACCEPTED;
    }

    tlm_sync_enum nb_transport_fw( TRANS& trans,
PHASE& phase, sc_core::sc_time& t )
```

```
     {

    //----------------------
    /* Приближенно-временный стиль кодирования,
использующий обратный путь */
    payload_event_queue->notify( trans, phase, t );
    trans.acquire();
    // Измените аргументы фазы и времени
     phase = END_REQ;
     t = t + accept_delay;
    return TLM_UPDATED;
    }

    //-------------------------------------
    /* вызов метода интерфейса b_transport, стиль
кодирования свободно-временной */
    initialize_transaction( T1 );
    socket->b_transport( T1, t ); // t may increase
     process_response( T1 );

    initialize_transaction( T2 );
    socket->b_transport( T2, t ); /* t может
увеличиться*/
     process_response( T2 );

    initialize_transaction( T2 );
    socket->b_transport( T2, t ); /* t может
увеличиться*/
    process_response( T2 );

    /* Инициатор может синхронизироваться после
каждой транзакции или после серии транзакций */
     quantum_keeper->set( t );
    if ( quantum_keeper->need_sync() )
    quantum_keeper->sync();


    // ----------------------------------------
    /* вызов метода интерфейса nb_transport,
приближенно-временной стиль кодирования */
    initialize_transaction( T3 );
    status = socket->nb_transport_fw( T3, phase, t );
```

```
    if ( status == TLM_ACCEPTED )
    {
    /* Нет действий, но ожидайте входящего вызова
метода nb_transport_bw */
     }
    else if ( status == TLM_UPDATED ) /* Используется
обратный путь*/
    {
    payload_event_queue->notify( T3, phase, t );
     }
    else if ( status == TLM_COMPLETED ) /* Раннее
завершение */
    {
    /* Аннотации времени могут быть учтены одним из
нескольких способов */
    // Либо (1), ожидая, как показано здесь
     wait ( t );
     process_response( T3 );
    // или (2) путем создания уведомления о событии
    response_event.notify( t );
     / * или (3) путем передачи следующего вызова
метода транспорта (код здесь не показан) */
    }
```

5.18. Интерфейс прямой памяти

Интерфейс прямой памяти или DMI предоставляет средство, с помощью которого инициатор может получить прямой доступ к области памяти, принадлежащей цели, после чего обращается к этой памяти с помощью прямого указателя, а не через интерфейс передачи. DMI предлагает большой потенциальный рост скорости моделирования для доступа к памяти между инициатором и целевым объектом, поскольку после его установки он может обходить обычный путь из нескольких вызовов **b_transport** или **nb_transport** от инициатора через компоненты межсоединений к цели. Существует два интерфейса прямой памяти: один для вызовов по прямому пути от инициатора к цели и второй для вызовов на обратном пути от цели до инициатора. Прямой путь используется для запроса определенного режима доступа DMI (например, чтения или записи) к данному адресу и возвращает ссылку на дескриптор DMI типа `tlm_dmi`, который содержит границы области DMI. Обратный путь используется целью для отклонения указателей DMI, ранее установленных с использованием прямого пути. Прямой и обратный пути могут проходить через нуль, один или многие компоненты межсоединений, но должны быть идентичны

прямым и обратным маршрутам для соответствующих транспортных вызовов через одни и те же сокеты.

Указатель DMI запрашивается путем передачи транзакции по прямому пути. Тип транзакции по умолчанию DMI - `tlm_generic_payload`, где используются только атрибуты команды и адреса объекта транзакции. DMI следует тому же подходу к расширению, что и транспортный интерфейс, то есть запрос DMI может содержать неосведомленные расширения, но для любого неосведомленного или обязательного расширения требуется определение нового класса признаков протокола (определение новых признаков протокола типа `class`, содержащий `typedef` для `tlm_generic_payload`).

Дескриптор DMI возвращает значения задержки для использования инициатором и, таким образом, обеспечивает достаточную точность синхронизации для свободно-временного моделирования. Указатели DMI могут использоваться для отладки, но сам интерфейс отладки обычно достаточен, поскольку трафик отладки обычно является легким и обычно доминирует при вводе-выводе, а не в доступе к памяти. Отладочные транзакции обычно не относятся к критическому пути для скорости моделирования. Если для отладки использовались указатели DMI, значения задержки должны игнорироваться.

5.18.1. Определение класса DMI

Для опреления класса DMI используют следующий синтаксис:

Листинг 5.14

```
    namespace tlm {

    class tlm_dmi
    {
    public:
    tlm_dmi() { init(); }

    void init();
    enum dmi_access_e {
    DMI_ACCESS_NONE = 0x00,
    DMI_ACCESS_READ = 0x01,
    DMI_ACCESS_WRITE = 0x02,
    DMI_ACCESS_READ_WRITE = DMI_ACCESS_READ |
DMI_ACCESS_WRITE
    };

    unsigned char* get_dmi_ptr() const;
    sc_dt::uint64 get_start_address() const;
```

```
    sc_dt::uint64 get_end_address() const;
    sc_core::sc_time get_read_latency() const;
     sc_core::sc_time get_write_latency() const;
    dmi_access_e get_granted_access() const;
    bool is_none_allowed() const;
    bool is_read_allowed() const;
    bool is_write_allowed() const;
    bool is_read_write_allowed() const;

    void set_dmi_ptr(unsigned char* p);
    void set_start_address(sc_dt::uint64 addr);
    void set_end_address(sc_dt::uint64 addr);
     void set_read_latency(sc_core::sc_time t);
     void set_write_latency(sc_core::sc_time t);
     void set_granted_access(dmi_access_e t);
    void allow_none(); void allow_read();
    void allow_write(); void allow_read_write();
    };

    template <typename TRANS = tlm_generic_payload>
    class tlm_fw_direct_mem_if : public virtual
sc_core::sc_interface
    {
    public:
    virtual bool get_direct_mem_ptr(TRANS& trans,
tlm_dmi& dmi_data) = 0;
    };
    class tlm_bw_direct_mem_if : public virtual
sc_core::sc_interface
    {
    public:
    virtual void
invalidate_direct_mem_ptr(sc_dt::uint64 start_range,
sc_dt::uint64 end_range) = 0;
    };
    } // namespace tlm
```

Подробную информацию об использовании и кодировании интерфейса DMI, о шаблонах аргументов и **tlm_generic_payload class** можно получить в OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL.

Там же описаны следующие темы:

Debug transport interface - Отладка транспортного интерфейса

Global quantum - Глобальное квантование
Combined interface and sockets - Комбинированный интерфейс и сокеты
Generic payload - Общая полезная нагрузка
Base protocol and phase - Базовый протокол и фаза
Utilities - Утилиты

5.19. Примеры программ с использованием TLM-2.0

5.19.1. Пример блокирующего интерфейса

Исходный код этого первого примера вы найдете в файле `tlm2_getting_started_1.cpp` [18] от компании Doulos.com (http://www.doulos.com)

Эта программа показывает общую полезную нагрузку, сокеты и блокирующий транспортный интерфейс. Показывает обязанности инициатора и цели в отношении общей полезной нагрузки. Имеет только фиктивные реализации интерфейсов прямой памяти и отладки транзакций. Не отображает неблокирующий транспортный интерфейс. Для программы необходимо ввести в Eclipse дополнительное вложение: `C:/TLM/Include/tlm`, в котором собраны заголовочные файлы и утилиты TLM-2.0 из SystemC-2.3.1. Кроме того, заголовочные файлы, использованные в программе включены в папку src (рис. 5.12).

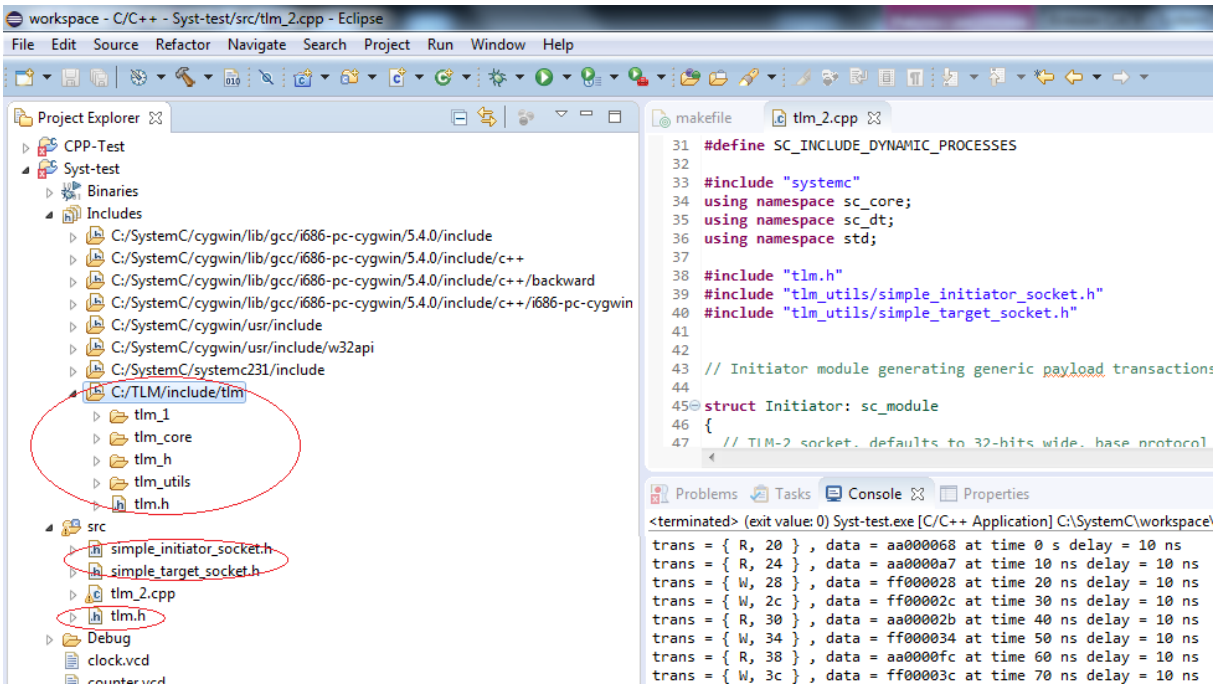


Рис. 5.12. Решение программы tlm2_getting_started_1.cpp в Eclipse

Листинг 5.15

Программа tlm2_getting_started_1.cpp

```
// Filename: tlm2_getting_started_1.cpp


#define SC_INCLUDE_DYNAMIC_PROCESSES

#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using namespace std;

#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"


/* Модуль инициатора, генерирующий общие
транзакции полезной нагрузки */

struct Initiator: sc_module
{
  /* TLM-2 сокет, по умолчанию 32-битный, базовый
протокол */
  tlm_utils::simple_initiator_socket<Initiator>
socket;

  SC_CTOR(Initiator)
  : socket("socket")  /* Построить и назвать
сокет */
  {
    SC_THREAD(thread_process);
  }

  void thread_process()
  {
    /* TLM-2 общая транзакция полезной нагрузки,
повторно используемая для вызовов b_transport */
    tlm::tlm_generic_payload* trans = new
tlm::tlm_generic_payload;
```

```
        sc_time delay = sc_time(10, SC_NS);


        /* Генерировать случайную последовательность
чтения и записи */
        for (int i = 32; i < 96; i += 4)
        {

          tlm::tlm_command cmd =
static_cast<tlm::tlm_command>(rand() % 2);
          if (cmd == tlm::TLM_WRITE_COMMAND) data =
0xFF000000 | i;

          /* Инициализировать 8 из 10 атрибутов,
byte_enable_length и расширения, которые не
используются */
          trans->set_command( cmd );
          trans->set_address( i );
          trans->set_data_ptr(
reinterpret_cast<unsigned char*>(&data) );
          trans->set_data_length( 4 );
          trans->set_streaming_width( 4 ); /*=
data_length для указания отсутствия потоковой
передачи */
          trans->set_byte_enable_ptr( 0 ); / * 0
указывает на неиспользованные * /
          trans->set_dmi_allowed( false ); / *
Обязательное начальное значение * /
    trans->set_response_status(
tlm::TLM_INCOMPLETE_RESPONSE ); / * Обязательное
начальное значение * /


          socket->b_transport( *trans, delay );
    // Блокировка транспортного вызова

          /* Инициатор обязан проверить статус ответа
и задержку */
          if ( trans->is_response_error() )
            SC_REPORT_ERROR("TLM-2", "Response error
from b_transport");
```

```
          cout << "trans = { " << (cmd ? 'W' : 'R')
<< ", " << hex << i
               << " } , data = " << hex << data << "
at time " << sc_time_stamp()
               << " delay = " << delay << endl;

          /* Реализовать задержку, аннотированную на
транспортный вызов */
          wait(delay);
        }
      }

      /* Внутренний буфер данных, используемый
инициатором с общей полезной нагрузкой */
      int data;
    };


    /* Целевой модуль, представляющий собой простую
память*/

    struct Memory: sc_module
    {
      /* TLM-2 сокет, по умолчанию 32-битный, базовый
протокол */
      tlm_utils::simple_target_socket<Memory> socket;

      enum { SIZE = 256 };

      SC_CTOR(Memory)
      : socket("socket")
      {
        /* Регистрация обратного вызова для входящего
вызова метода интерфейса b_transport */
        socket.register_b_transport(this,
&Memory::b_transport);

        /* Инициализировать память со случайными
данными*/
        for (int i = 0; i < SIZE; i++)
          mem[i] = 0xAA000000 | (rand() % 256);
      }
```

```
      // Метод блокирующего транспорта TLM-2
      virtual void b_transport(
tlm::tlm_generic_payload& trans, sc_time& delay )
      {
        tlm::tlm_command cmd = trans.get_command();
        sc_dt::uint64    adr = trans.get_address() /
4;
        unsigned char*   ptr = trans.get_data_ptr();
        unsigned int     len =
trans.get_data_length();
        unsigned char*   byt =
trans.get_byte_enable_ptr();
        unsigned int     wid =
trans.get_streaming_width();

    /* Обязательно проверять диапазон адресов и
проверять наличие неподдерживаемых функций, то есть
байт включений, потоковую передачу и пакеты*/
     // Можно игнорировать подсказку и расширения DMI
     / * Использование обработчика отчета SystemC
является приемлемым способом сигнализации об ошибке
*/

        if (adr >= sc_dt::uint64(SIZE) || byt != 0 ||
len > 4 || wid < len)
          SC_REPORT_ERROR("TLM-2", "Target does not
support given generic payload transaction");

        /* Обязательно выполнять команды чтения и
записи */
        if ( cmd == tlm::TLM_READ_COMMAND )
          memcpy(ptr, &mem[adr], len);
        else if ( cmd == tlm::TLM_WRITE_COMMAND )
          memcpy(&mem[adr], ptr, len);

        /* Обязательно установить статус ответа,
чтобы указать успешное завершение */
        trans.set_response_status(
tlm::TLM_OK_RESPONSE );
      }
```

```
  int mem[SIZE];
};


SC_MODULE(Top)
{
  Initiator *initiator;
  Memory    *memory;

  SC_CTOR(Top)
  {
    // Создание компонентов
    initiator = new Initiator("initiator");
    memory    = new Memory   ("memory");


    /* Один инициатор связан непосредственно с
одной целью без промежуточной шины */

    /* Связать разъем инициатора с целевым
сокетом*/
    initiator->socket.bind( memory->socket );
  }
};



int sc_main(int argc, char* argv[])
{
  Top top("top");
  sc_start();
  return 0;
}
```

Результаты моделирования

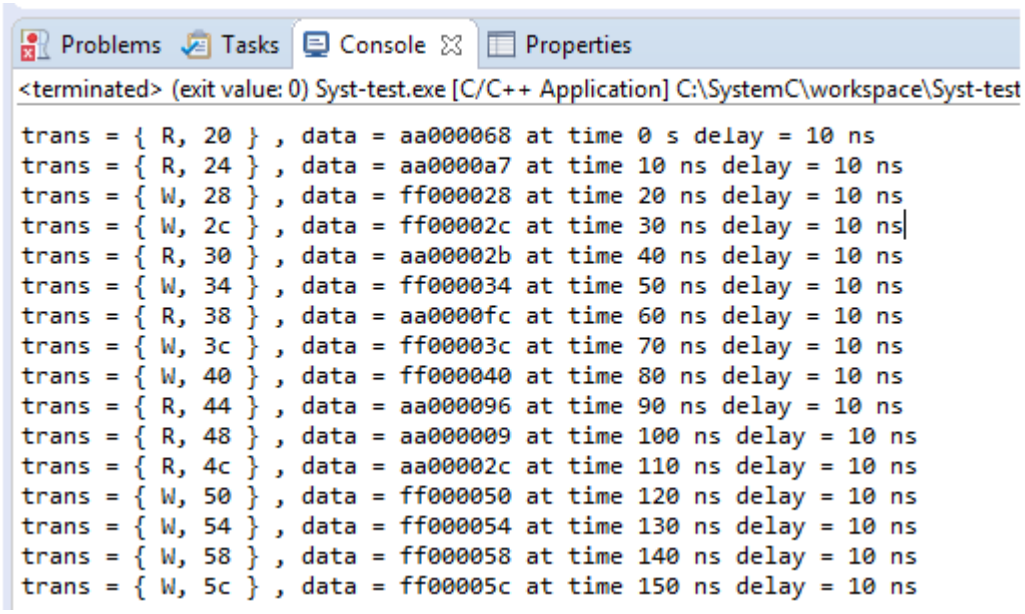


Рис. 5.13. Результаты моделирования

5.19.2. Пример неблокирующего интерфейса `at_1_phase`

Рассмотрим в качестве примера использования TLM-2.0 программу, приведенную в примерах SystemC-2.3.1. Это пример системы AT – (Approximately Timed – Приближенно-временной стиль).

Цель состоит в том, чтобы проиллюстрировать:

- Применение TLM 2.0 в реальной системе;

-Аннотированную неблокирующую (NB) опцию неблокирующего стиля;

NB аннотированное время, которое называется «1 фазой»;

Простейшую версию неблокирующего / AT стиля.

Возможные применения:

- исследование архитектуры;

- предварительная разработка программного обеспечения

Блок-схема примера показана на рис. 5.14.

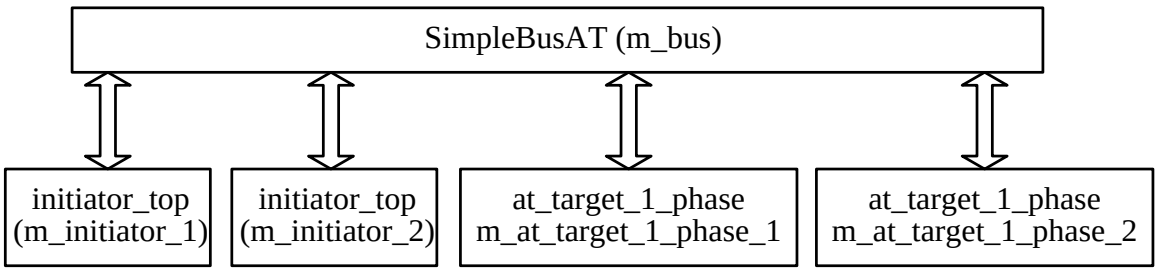


Рис. 5.14. Блок-схема примера

Модель имеет два инициатора, две цели и маршрутизатор. Цели являются разными фазовыми объектами (1 и 2).

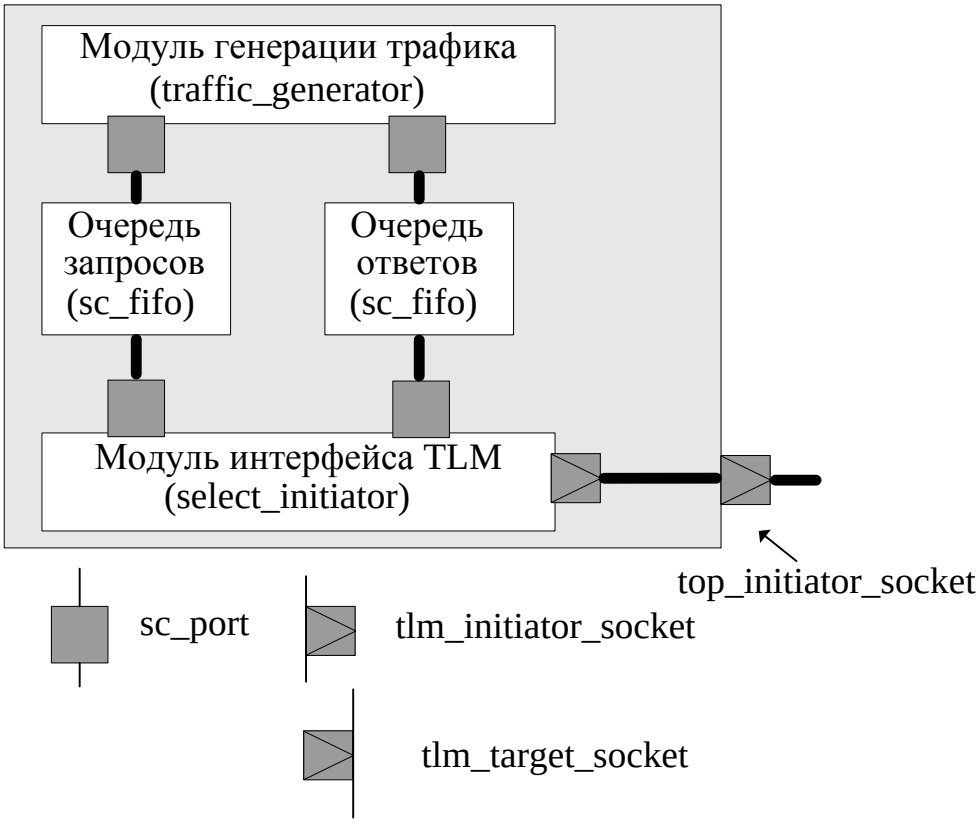


Рис. 5.15. Модуль инициатора

Инициатор содержит модель генератора передачи (Traffic Generator Module), очередь запросов (Request Queue) и очередь ответов (Response Queue), выполненные как sc_fifo, и интерфейсный TLM модуль с выбором инициатора.

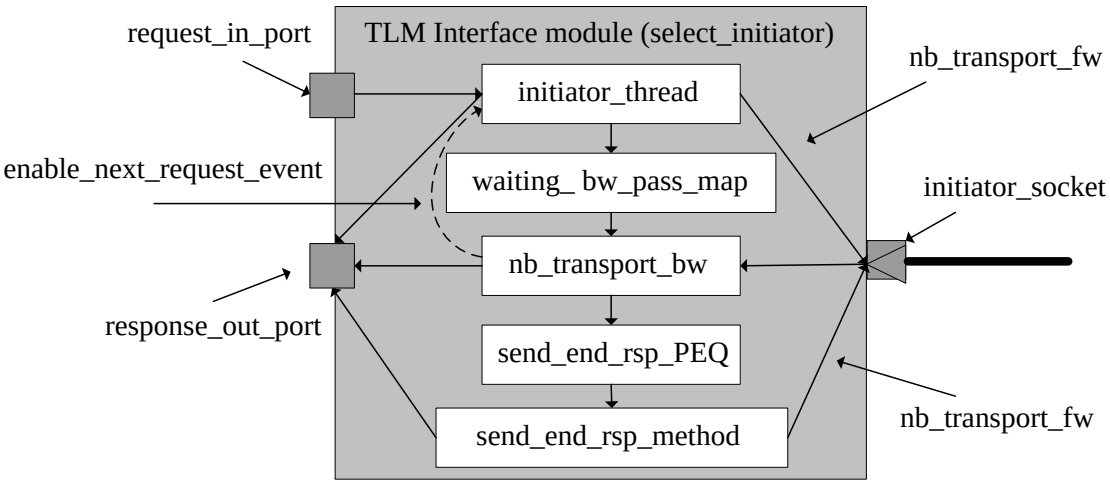


Рис. 5.16. Модуль интерфейса

Модуль TLM интерфейса содержит входной порт запросов, выходной порт ответов, сокет инициатора. Входному запрос инициирует поток инициатора и вызывает метод **nb_transport_fw**, который передается на сокет инициатора, на выходной порт ответа и на метод ожидания по карте прямой передачи. С сокета модуля интерфейса поступает ответный сигнал и запускает метод обратного интерфейса nb_transport_bw, с которого выдается сигнал на выходной порт ответа, разрешение на следующий запрос и на метод send_end_rsp_method, посылающий на сокет и на выходной порт сигнал конца ответа.

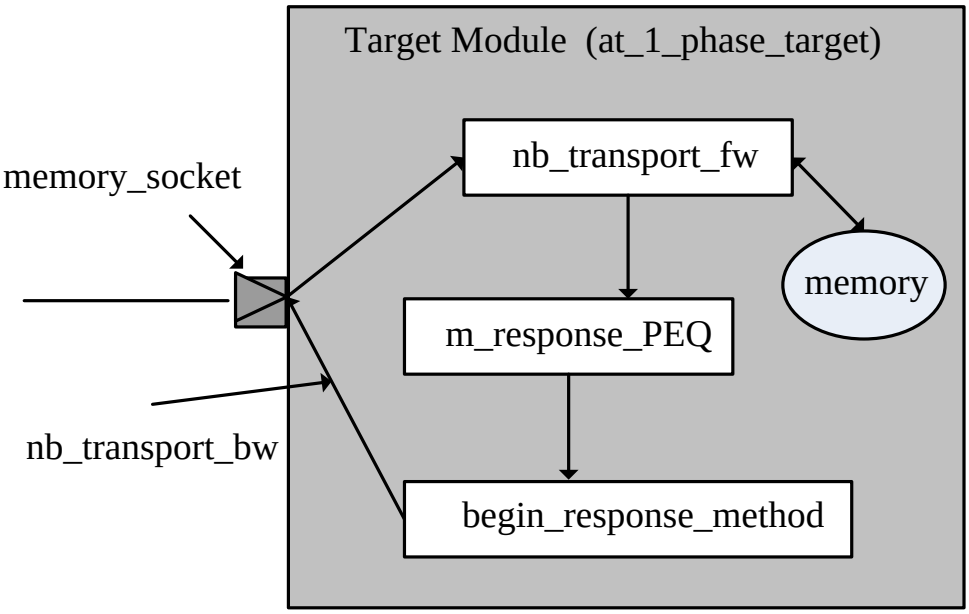


Рис. 5.17. Модуль цели

Модуль цели имеет сокет, через который по интерфейсу nb_transport_fw принятые данные передаются в память и формируется ответ для посылки инициатору.

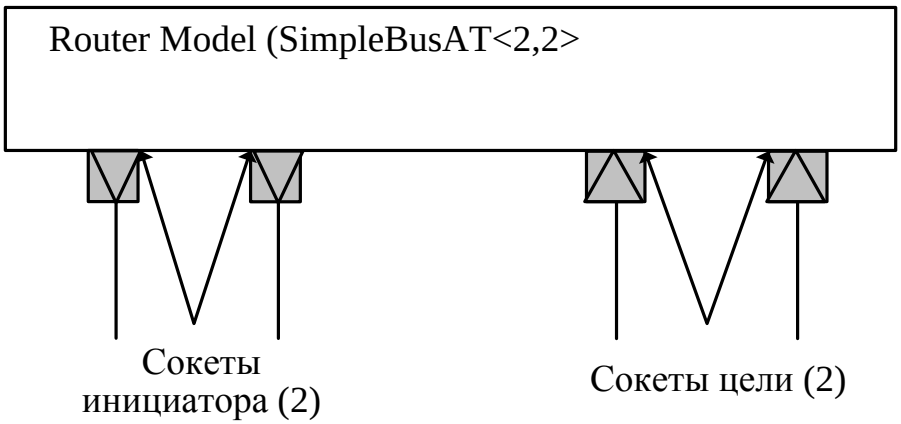


Рис. 5.18. Компонент маршрутизатор

Маршрутизатор построен по принципу простой шины с приближенным временем и содержит сокеты инициаторов и целей.

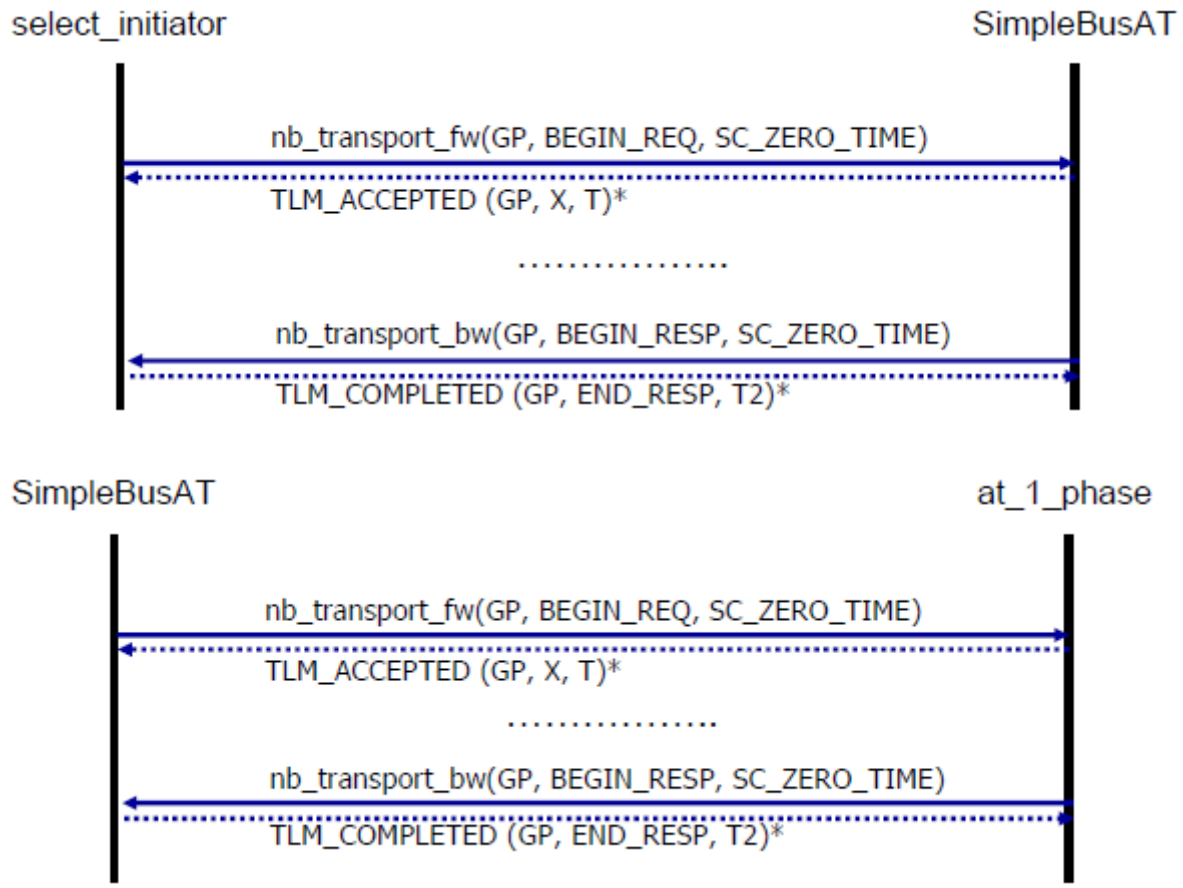


Рис. 5.19. Ожидаемая временная диаграмма

Этот пример из SystemC-2.3.1 рекомендуется компилировать и моделировать в Linux или в Microsoft Visual Studio.

Мы выполнили это в MSVC следующим образом.

1. Из папки `SystemC-2.3.1/examples/tlm/at_1_phase/build-windows` запустили файл `at_1_phase.sln`. В результате открылось рабочее поле MSVC с проектом `at_1_phase`.

2. Выбрали ‘Property Manager’ из меню ‘View’.

Для проекта `at_1_phase` установили Debug | Win32 и выбрали ‘systemc’

Выбрали ‘Properties’ из меню ‘View’ .

Выбрали ‘User Macros’ на вкладке ‘Common Properties’.

Далее потребовалось выполнить обновление вложенных файлов. Практически это сводится к тому, что при очередной компиляции проекта

выявляются ошибки, связанные с отсутствием нужных файлов или пути к ним. В этом случае в папках SystemC-2.3.1 надо отыскать потерянный файл и прописать путь к его папке в свойствах проекта VC++Directories/ Include Directories (Рис. 5.20).

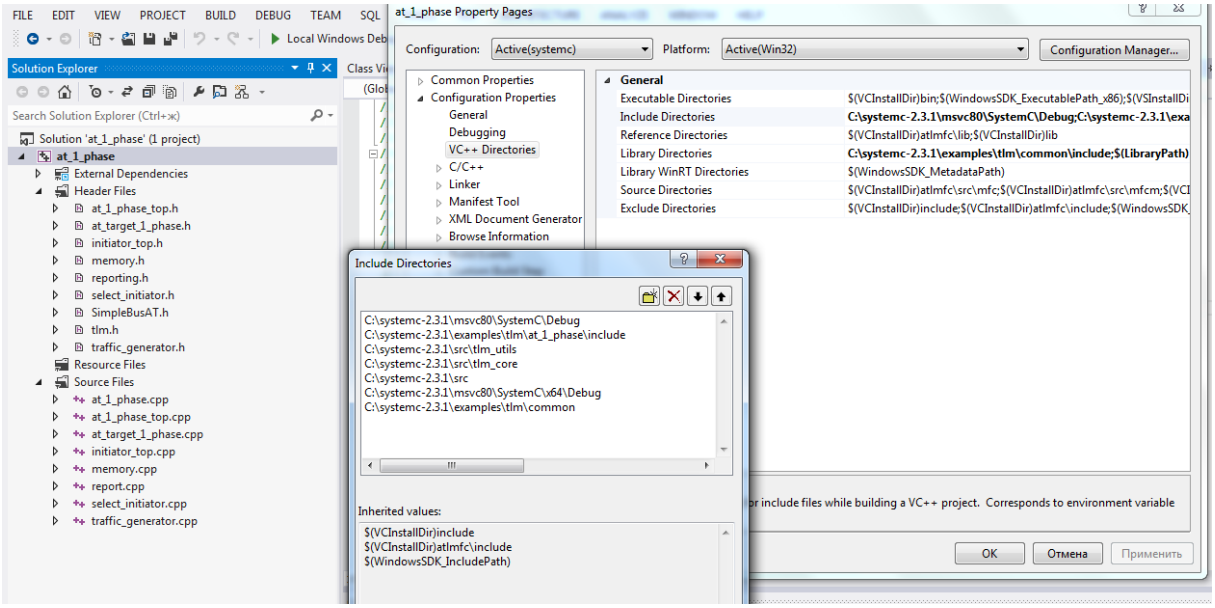


Рис. 5.20. Вложенные директории

Для «перестраховки» мы сделали это и для C++/Additional Include Directories (рис. 5.21).

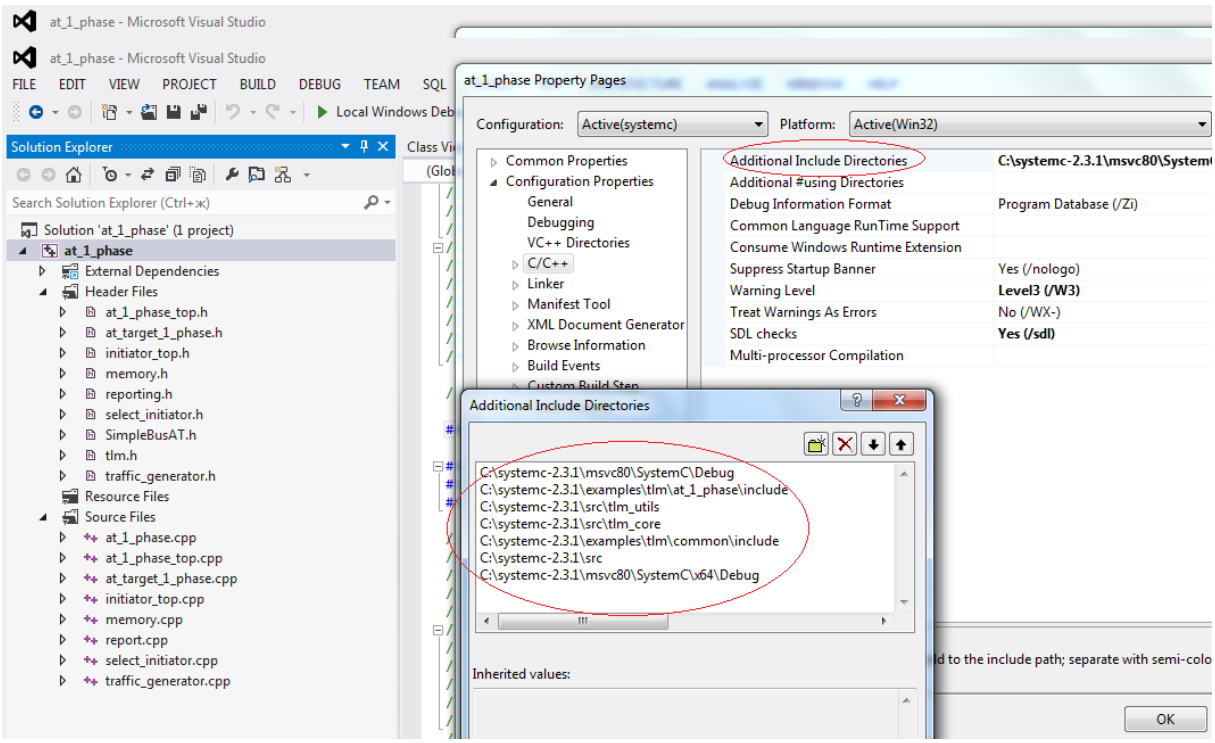


Рис. 5.21. Дополнительные вложенные директории

После этого осталась одна ошибка, связанная с библиотекой systemc.lib (рис. 5.22).

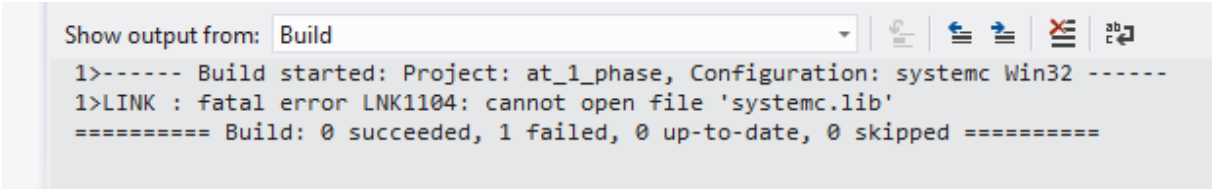
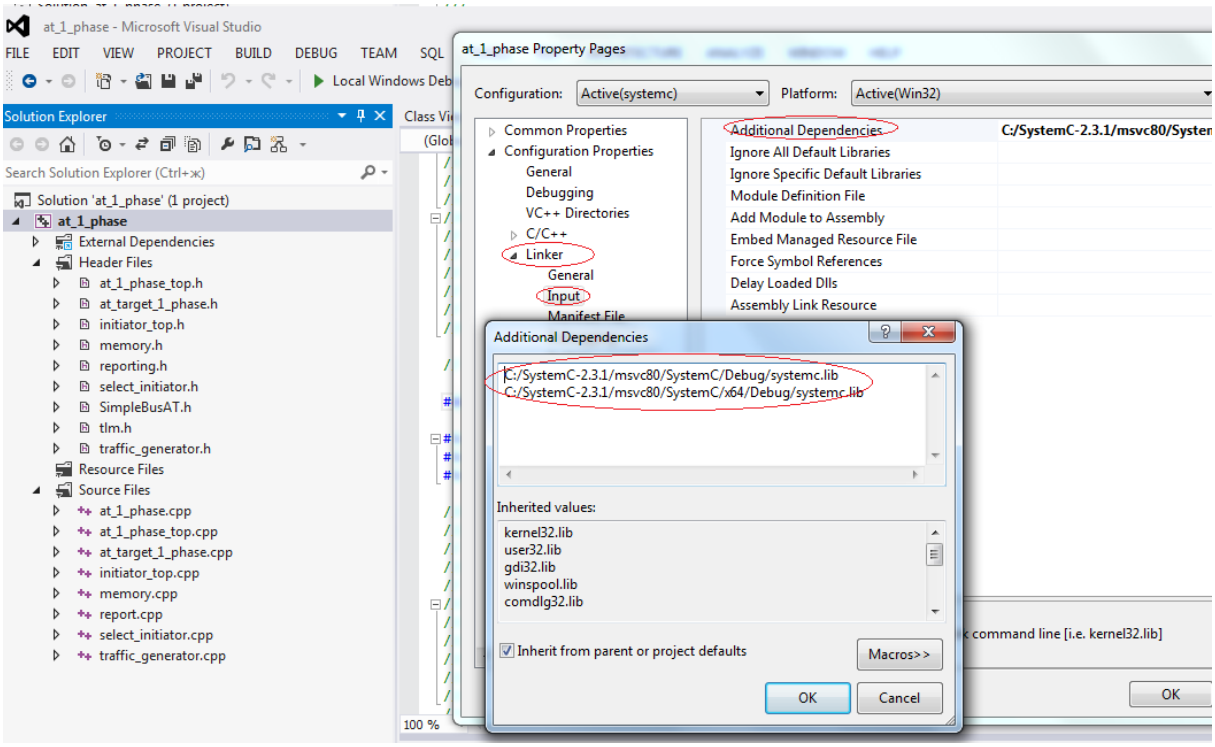


Рис. 5.22. Нет доступа к библиотеке systemc.lib

Мы сделали добавление в линкер: выбираем Linker-Input-Additional Dependecies и добавляем (рис. 5.23):

```
C:/SystemC-2.3.1/msvc80/SystemC/Debug/systemc.lib;
C:/SystemC-2.3.1/msvc80/SystemC/x64/Debug/systemc.lib.
```

После этого программа была успешно скомпилирована (рис. 5.24).



Ри. 5.23. Добавление в Линкер

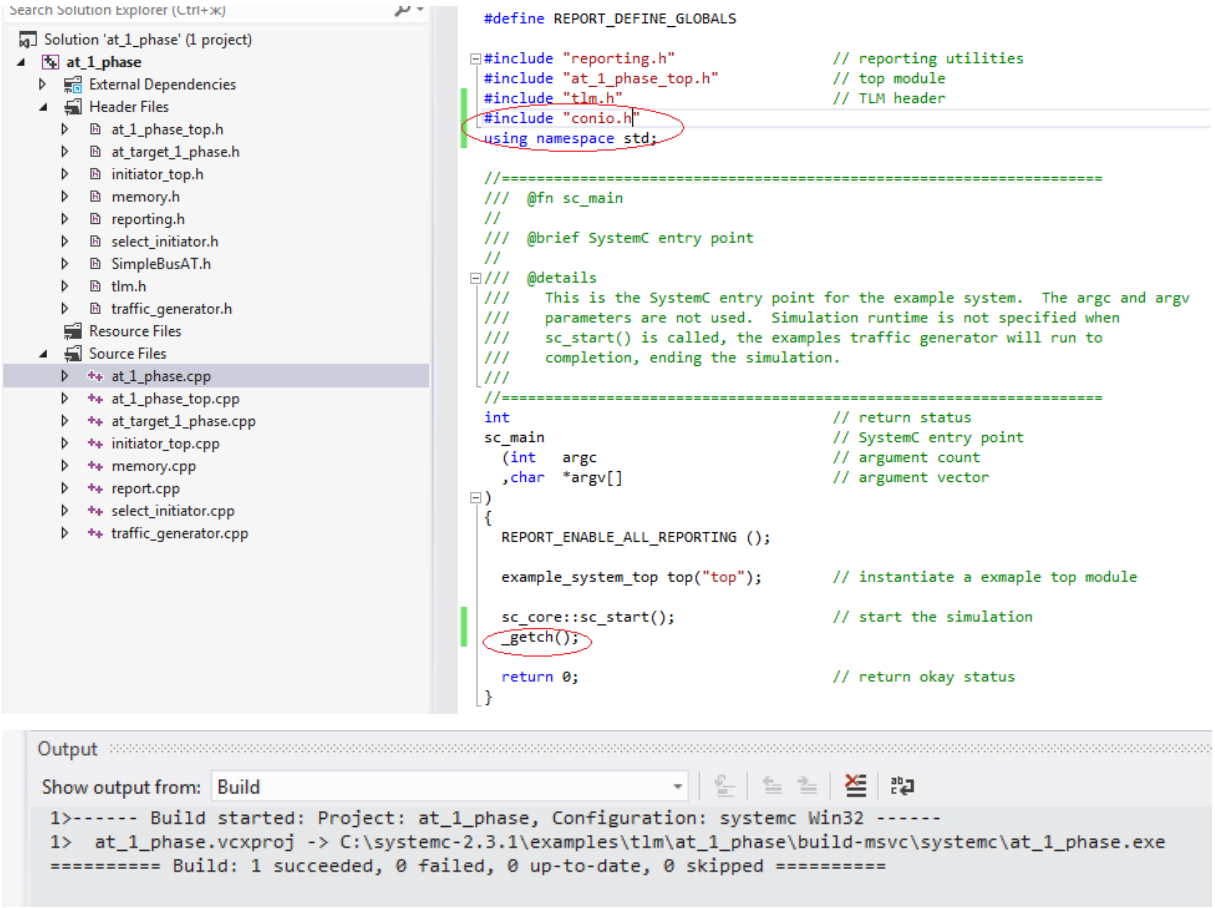


Рис. 5.24. Компиляция программы

Для вывода результатов в программу добавлены: `#include “conio.h”, using namespace std; _getch().`

Большое количество и объем заголовочных и исполняемых файлов не позволяет привести полные листинги программ. Вы можете найти эти файлы в примерах SystemC-2.3.1 и поработать с ними при запуске программы.

Проводим основные заголовочные и исполняемые файлы.

Заголовочные файлы

Листинг 5.16

Заголовочный файл example_system_top.h (at_1_phase_top.h)

```
    /*  Этот класс создает компоненты, которые
составляют TLM2 */

    #ifndef __EXAMPLE_SYSTEM_TOP_H__
    #define __EXAMPLE_SYSTEM_TOP_H__
```

```
    #include "reporting.h"  // общий код отчетности
    #include "at_target_1_phase.h"// по цели памяти
    #include "initiator_top.h" /* инициатор
абстракции процессора */
    #include "models/SimpleBusAT.h"/* Реализация шины
/ маршрутизатора */

    class example_system_top
    : public sc_core::sc_module //  Основной класс SC
    {
    public:

    // Конструктор

      example_system_top
      ( sc_core::sc_module_name name);

    // Переменные члены
     private:
      SimpleBusAT<2, 2>       m_bus;   // простая шина
      at_target_1_phase       m_at_target_1_phase_1;
    //экземпляр  цели 1
      at_target_1_phase       m_at_target_1_phase_2;
     //экземпляр цели 2
      initiator_top           m_initiator_1;
     //экземпляр инициатора 1
      initiator_top           m_initiator_2;
              // экземпляр инициатора 2
    };

    #endif /* __EXAMPLE_SYSTEM_TOP_H__ */
```

Листинг 5.17

Заголовочный файл initiator_top.h

```
    /*Верхний модуль-инициатора содержит генератор
трафика и примерный модуль-инициатор tlm */
    #ifndef __INITIATOR_TOP_H__
    #define __INITIATOR_TOP_H__

    #include "tlm.h"     // Заголовки TLM
    #include "select_initiator.h"  // AT инициатор
```

```
    #include "traffic_generator.h" /* Генератор
трафика*/

    class initiator_top    : public
sc_core::sc_module
      , virtual public tlm::tlm_bw_transport_if<>
     // обратный неблокирующий интерфейс
    {
    //Методы - члены

=======================================================

      public:

    /*  Верхний модуль инициатора содержит генератор
трафика и пример уникального модуля инициатора */
      initiator_top
      ( sc_core::sc_module_name name
      , const unsigned int  ID
      , sc_dt::uint64       base_address_1
      , sc_dt::uint64       base_address_2
      , unsigned int        active_txn_count
            );

    private:

    /* Не реализовано для этого примера, но требуется
сокет инициатора */
      void

      invalidate_direct_mem_ptr
      ( sc_dt::uint64      start_range
      , sc_dt::uint64      end_range
      );

    /* Не реализовано для этого примера, но требуется
сокет инициатора */
      tlm::tlm_sync_enum
      nb_transport_bw

      ( tlm::tlm_generic_payload  &payload
      , tlm::tlm_phase            &phase
```

```
      , sc_core::sc_time          &delta
      );

    // Члены переменные / объекты
=========================================

      public:

      tlm::tlm_initiator_socket<> initiator_socket;
      private:

      typedef tlm::tlm_generic_payload  *gp_ptr;
      sc_core::sc_fifo <gp_ptr>  m_request_fifo;
      sc_core::sc_fifo <gp_ptr>  m_response_fifo;
      const unsigned int         m_ID;
       bool
       m_enable_target_tracking;

      select_initiator           m_initiator;
      traffic_generator          m_traffic_gen;
    };

    #endif /* __INITIATOR_TOP_H__ */
```

Исполняемые файлы

Листинг 5.18

File_example_main.cpp (at_1_phase.cpp)

```
    /* Определите REPORT_DEFINE_GLOBALS только в
одном месте */

    #define REPORT_DEFINE_GLOBALS

    #include "reporting.h"   /* утилиты для
отчетности*/
    #include "at_1_phase_top.h"  // Верхний модуль
    #include "tlm.h"             // Заголовок TLM
    #include "conio.h"
    using namespace std;
```

```
int               // статус возврата
sc_main           // Входная точка SystemC
  (int   argc     // количество аргументов
  ,char  *argv[] // вектор аргумента
)
{
  REPORT_ENABLE_ALL_REPORTING ();

  example_system_top top("top");
// создать экземпляр верхнего модуля

  sc_core::sc_start();   // начать симуляцию
  _getch();

  return 0;              // возврат okay статуса
}
```

Листинг 5.19

File_example_system_top.cpp (at_1_phase_top.cpp)

```
    #include "at_1_phase_top.h"     /* пример
верхнего заголовка системы */
    /* Метод конструктора вызывает методы связывания
для подключения компонентов примера*/
    example_system_top::example_system_top
    ( sc_core::sc_module_name name
    )
      : sc_core::sc_module  // Инициировать SC base
        ( name
        )
      , m_bus             // Инициировать Simple Bus
        ( "m_bus"
        )
      , m_at_target_1_phase_1
    // Init intance 1 of AT target
        ( "m_at_target_1_phase_1"
    // module name
        , 201
    // 1st Target ID is 201
        , "memory_socket_1"     // имя сокета
        , 4*1024           // размер памяти (байты)
        , 4                // ширина памяти (байты)
```

```
        , sc_core::sc_time(10, sc_core::SC_NS)
    // принять задержку
        , sc_core::sc_time(50, sc_core::SC_NS)
    // считывать задержку ответа
        , sc_core::sc_time(30, sc_core::SC_NS)
    // задержка ответа на запись
        )
      , m_at_target_1_phase_2
    // Идентификатор экземпляра 2 цели AT
        ( "m_at_target_1_phase_2"  // Имя модуля
        , 202
    // 2nd Target ID is 202
        , "memory_socket_1"        // Имя сокета
        , 4*1024           // размер памяти (байты)
        , 4                // ширина памяти (байты)
        , sc_core::sc_time(10, sc_core::SC_NS)
    // принять задержку
        , sc_core::sc_time(50, sc_core::SC_NS)
    // считывать задержку ответа
        , sc_core::sc_time(30, sc_core::SC_NS)
    // задержка ответа на запись
        )
      , m_initiator_1
    // Инициировать экземпляр 1 инициатора AT
        ( "m_initiator_1"       // имя модуля
        , 101        /* Идентификатор первого
инициатора - 101*/
        , 0x0000000000000100   /* первый базовый
адрес*/
    , 10000100               /* второй базовый
ардес*/
        , 2                 // активные транзакции
        )
      , m_initiator_2   // Инициирующий инициатор 2
        ( "m_initiator_2"     // Имя модуля
        , 102          /* Идентификатор второго
инициатора - 102*/
        , 0x0000000000000200
    // первый базовый адрес
        , 0x0000000010000200
    // второй базовый адрес
        , 2              // активные транзакции
```

```
        )
    {
      /* Связывать инициаторы TLM2 с целевыми
сокетами TLM2 на SimpleBus */

m_initiator_1.initiator_socket(m_bus.target_socket[0]
);

m_initiator_2.initiator_socket(m_bus.target_socket[1]
);

      / * связывать цели TLM2 с сокетами инициатора
TLM2 на SimpleBus * /

m_bus.initiator_socket[0](m_at_target_1_phase_1.m_mem
ory_socket);

m_bus.initiator_socket[1](m_at_target_1_phase_2.m_mem
ory_socket);
    }
```

Листинг 5.20

File_initiator_top.cpp

```
    #include "initiator_top.h"
// Главный генератор трафика и инициатор
    #include "reporting.h"
// Отчетность макро помощников

    static const char *filename =
"initiator_top.cpp";
    // Конструктор

    initiator_top::initiator_top
    ( sc_core::sc_module_name name
    , const unsigned int    ID
    , sc_dt::uint64         base_address_1
    , sc_dt::uint64         base_address_2
    , unsigned int          active_txn_count
    )
      :sc_module     (name) /* имя модуля для
верхнего инициатора */
```

```
      ,initiator_socket ("at_initiator_socket")
    // TLM сокет

      ,m_ID             (ID)       // ID инициатора

      ,m_initiator          // Инициирующий инициатор
        ("m_initiator_1_phase"
        ,ID
        ,sc_core::sc_time(7, sc_core::SC_NS)
    // установить задержку инициатора end rsp
        )

      ,m_traffic_gen      /* Инициировать генератор
трафика*/
        ("m_traffic_gen"
        ,ID
        ,base_address_1      // первый базовый адрес
        ,base_address_2     // второй базовый адрес
        ,active_txn_count  /* Максимальные активные
транзакции*/
        )

    {
      /* Свяжите порты с m_request_fifo между
m_initiator и m_traffic_gen */
      m_traffic_gen.request_out_port
(m_request_fifo);
      m_initiator.request_in_port
(m_request_fifo);

      /* Свяжите порты с m_response_fifo между
m_initiator и m_traffic_gen */
      m_initiator.response_out_port
(m_response_fifo);
      m_traffic_gen.response_in_port
(m_response_fifo);

      /* Связывать инициатор-сокет с иерархическим
соединением инициатор-сокет */
      m_initiator.initiator_socket(initiator_socket);
    }
```

```
    /* В этом примере DMI не нет, поэтому не
используется */
    void
    initiator_top::invalidate_direct_mem_ptr

    ( sc_dt::uint64      start_range
     ,sc_dt::uint64      end_range
    )
    {
      std::ostringstream       msg;   /* сообщение
журнала*/
      msg.str ("");

      msg << "Initiator: " << m_ID << " Not
implemented";
      REPORT_ERROR(filename, __FUNCTION__,
msg.str());
    } // конец invalidate_direct_mem_ptr


    tlm::tlm_sync_enum
    initiator_top::nb_transport_bw

    ( tlm::tlm_generic_payload  &payload
     ,tlm::tlm_phase            &phase
     ,sc_core::sc_time          &delta
    )
    {
      std::ostringstream       msg;   /* сообщение
журнала*/
      msg.str ("");

      msg << "Initiator: " << m_ID
          << " Not implemented, for hierachical
connection of initiator socket";
      REPORT_ERROR(filename, __FUNCTION__,
msg.str());

      return tlm::TLM_COMPLETED;

     } // Конец nb_transport_bw
```

Результаты решения частично показаны на рис. 5.25.

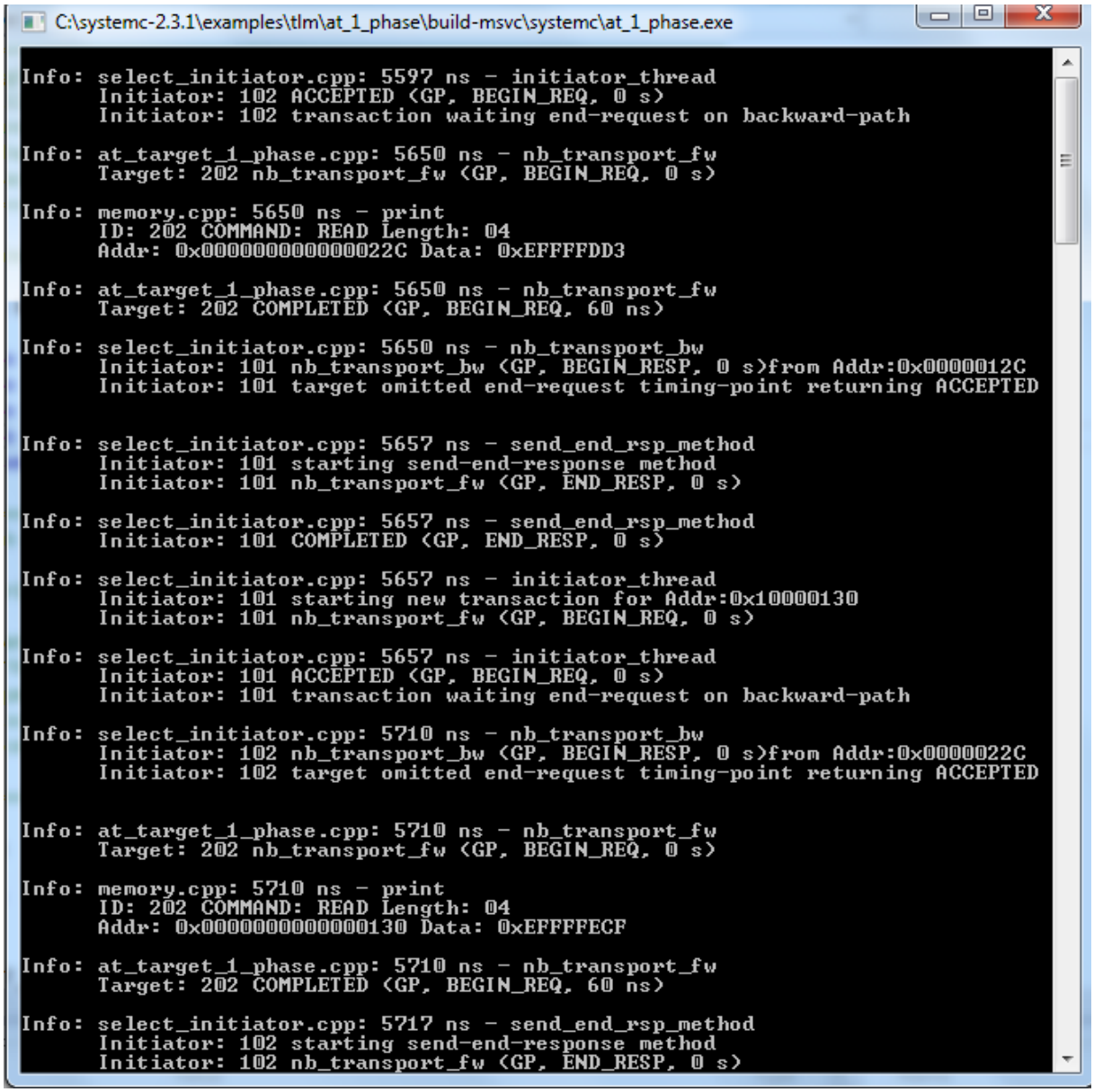


Рис. 5.25. Фрагмент результатов решения программы `at_1_phase`

5.20. Выводы: Основные характеристики TLM-2

1. Транспортные интерфейсы с аннотацией и фазами синхронизации.

2. Интерфейсы DMI и отладки.

3. Свободно-временной стиль кодирования и временная развязка для повышения скорость моделирования

4. Приближенно - временный стиль кодирования для точности синхронизации.

5. Сокеты для удобства и надежной проверки соединения.

6. Общая полезная нагрузка для моделирования шины с памятью.

7. Базовый протокол для взаимодействия между TL-моделями.

8. Расширение гибкости моделирования.

Что дает TLM-2,0 для разработчиков электронных устройств на системном уровне (ESL - Electronic system level) ?

1. Хорошо определенные варианты использования, стили моделирования и API TLM

- Модельная совместимость
- Доступность модели

2. Высокоскоростное моделирование для виртуальных платформ на базе SystemC

- временная развязка, прямой интерфейс памяти, синхронизация по запросам

3. Повторное использование модели для нескольких задач проектирования ESL

- Модели LT взаимодействуют и могут быть доработаны до моделей AT
- Модели LT и AT могут быть подключены к средства трансакций.

Приложение А

А.1. Краткое введение в язык С++

SystemC является надстройкой для языка программирования С++ и практически все программы SystemC содержат операторы, переменные, константы, систаксис языка С++. Поэтому необходимо знать краткие сведения о С++.

Язык программирования С++ на сегодняшний момент является доминирующим языком *объектно-ориентированного программирования*.

Являясь одним из самых популярных языков программирования, С++ широко используется для разработки программного обеспечения.

Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также развлекательных приложений. С++ оказал огромное влияние на другие языки программирования, в первую очередь на Java и C#.

Методология объектно-ориентированного программирования – это подход, использующий объектную декомпозицию, при которой статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

Исследования в области моделирования реальных систем привели к необходимости создания средств естественного описания сущностей, которые в них встречаются: объектов и событий. Позже оказалось, что такие концепции, как инкапсуляция (абстрактные типы данных), наследование и полиморфизм являются достаточно полезным дополнением к традиционному структурному программированию. Возможность их достаточно эффективной реализации привела к созданию широко распространенных в наши дни объектно-ориентированных языков.

Синтаксис и семантика

Вычислительная модель чистого объектно-ориентированного программировании поддерживает явно только одну операцию, которой является посылка объекту сообщения. Сообщения могут иметь параметры, являющиеся объектами. Само сообщение также является объектом.

Объект имеет набор обработчиков сообщений (набор методов). У объекта есть поля – персональные переменные для данного объекта, значениями которых являются ссылки на другие объекты. В одном из полей объекта хранится ссылка на объект-предок, которому переадресуются все сообщения, не обрабатываемые данным объектом. Структуры, описывающие обработку и переадресацию сообщений, обычно

выделяют в отдельный объект, называемый классом данного объекта. Сам объект называют экземпляром указанного класса.

В объектно-ориентированном программировании определяют три основных свойства, перечисленные ниже.

Инкапсуляция – это сокрытие информации и комбинирование данных и функций, которые аналогичны абстрактным типам данных.

Наследование – построение иерархии порожденных объектов с возможностью для каждого такого объекта, относящегося к иерархии, доступа к методам и данным всех порождающих объектов.

Полиморфизм (полиморфизм включения) – присваивание действию одного имени, которое затем разделяется вверх и вниз по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, подходящим именно ему. Таким образом, одно и то же имя дается разным действиям, способ исполнения которых задается в различных классах.

А.1.1.Происхождение языка С++

Язык С++ был разработан в начале 1980-х гг. Бьерном Страуструпом из компании AT&T Bell Laboratories. С++ основан на языке Си. С++ был задуман как язык Си с расширенными возможностями. Большая часть языка Си вошла в С++ как подмножество, поэтому многие программы на Си можно скомпилировать (т.е. превратить в набор низкоуровневых команд, которые компьютер может непосредственно выполнять) с помощью компилятора Си++. В языке С++ операцию инкремента (увеличение значения переменной на 1) обозначают «++». Отсюда пошло название расширенного языка С++. Стандартный ANSI Си++ гарантированно можно запустить на любом компьютере, у которого имеется компилятор ANSI Си++.

Разработку и отладку программ выполняют в среде разработки программ (Microsoft Visual Studio C++ , Eclipse CDT C++). Интегрированная среда разработки (IDE) включает в себя редактор, компилятор, компоновщик и отладчик.

Справочная информация и примеры программ получены из [31-33].

А.1.2. Первая программа в С++

Набираем код первой программы:

Пример А.1

```
#include <iostream>
#include <cstdlib> // для system
using namespace std;
```

```
    int main()
    {
        cout << "Hello, world!" << endl;
        system("pause"); // Только для тех, у кого MS
Visual Studio
        return 0;
    }
```

Описание синтаксиса

Директива **#include** используется для подключения других файлов в код. Строка #include <iostream>, будет заменена содержимым файла «iostream.h», который находится в стандартной библиотеке языка и отвечает за ввод и вывод данных на экран.

#include `<cstdlib>` подключает стандартную библиотеку языка C. Это подключение необходимо для работы функции system.

Содержимое третьей строки — `using namespace std;` указывает на то, что мы используем по умолчанию пространство имен с названием «`std`».

Пространство имён (*namespace*) — некоторое множество, под которым подразумевается модель, абстрактное хранилище или окружение, созданное для логической группировки уникальных идентификаторов (то есть имён).

Идентификатор, определённый в пространстве имён, ассоциируется с этим пространством. Один и тот же идентификатор может быть независимо определён в нескольких пространствах. Таким образом, значение, связанное с идентификатором, определённым в одном пространстве имён, может иметь (или не иметь) такое же значение, как и такой же идентификатор, определённый в другом пространстве. Языки с поддержкой пространств имён определяют правила, указывающие, к какому пространству имён принадлежит идентификатор (то есть его определение).

В языке C++ пространство имён определяется блоком инструкций:

```
namespace foo {
  int bar;
}
```

Внутри этого блока идентификаторы могут вызываться именно так, как они были объявлены. Но вне блока требуется указание имени пространства имён перед идентификатором. Например, вне `namespace foo` идентификатор `bar` должен указываться как `foo::bar`. C++

содержит некоторые другие конструкции, делающие подобные требования необязательными. Так, при добавлении строки

```
using namespace foo;
```

в код, указывать префикс `foo::` больше не требуется.

Стандартная библиотека шаблонов (STL) — это подмножество стандартной библиотеки C++ и содержит контейнеры, алгоритмы, итераторы, объекты-функции и т.д. Заголовочные файлы стандартной библиотеки C++ не имеют расширения «.h».

Стандартная библиотека C++ содержит последние расширения C++ стандарта ANSI (включая библиотеку стандартных шаблонов и новую библиотеку `iostream`). Она представляет собой набор файлов заголовков. В новых файлах заголовков отсутствует расширение H.

Все идентификаторы стандартной библиотеки C++ в процессе стандартизации были объединены в пространство имен **std**

Все то, что находится внутри фигурных скобок функции `int main() {}` будет автоматически выполняться после запуска программы.

Строка `cout << "Hello, world!" << endl;` говорит программе выводить сообщение с текстом **«Hello, world»** на экран.

Оператор `cout` предназначен для вывода текста на экран командной строки. После него ставятся две угловые кавычки `(<<).` Далее идет текст, который должен выводиться. Он помещается в двойные кавычки.

Оператор `cin` выполняет ввод данных в ячейку переменной.

Пример А.2

В этом примере выполнен ввод текущей даты и вывод на консоль.

```
#include <iostream.h>
int main()
{
int day_now, month_now, year_now;
cout << "Введите текущую дату и нажмите
ENTER.\n";
cin >> day_now;
cout << "Введите месяц и нажмите ENTER.\n";
cin >> month_now;
cout << "Введите год и нажмите ENTER.\n";
cin >> year_now;
cout << day_now <<”.”<< month_now <<”.”<<
year_now<< "\n"<<endl;
return 0;
}
```

Оператор `endl` переводит строку на уровень ниже.

Если в процессе выполнения произойдет какой-либо сбой, то будет сгенерирован код ошибки, отличный от нуля. Если же работа программы завершилась без сбоев, то код ошибки будет равен нулю. Команда `return 0` необходима для того, чтобы передать операционной системе сообщение об удачном завершении программы.

В конце каждой команды ставится **точка с запятой.**

Для проверки и отладки программы создаем C++ Project с среде Eclipse. Порядок установки и настройки среды Eclipse CDT был подробно описан в главе 2.

Создаем проект C++ Project с названием CPP-Test, тип проекта Hellow World C++Project , компилятор Cygwin GCC

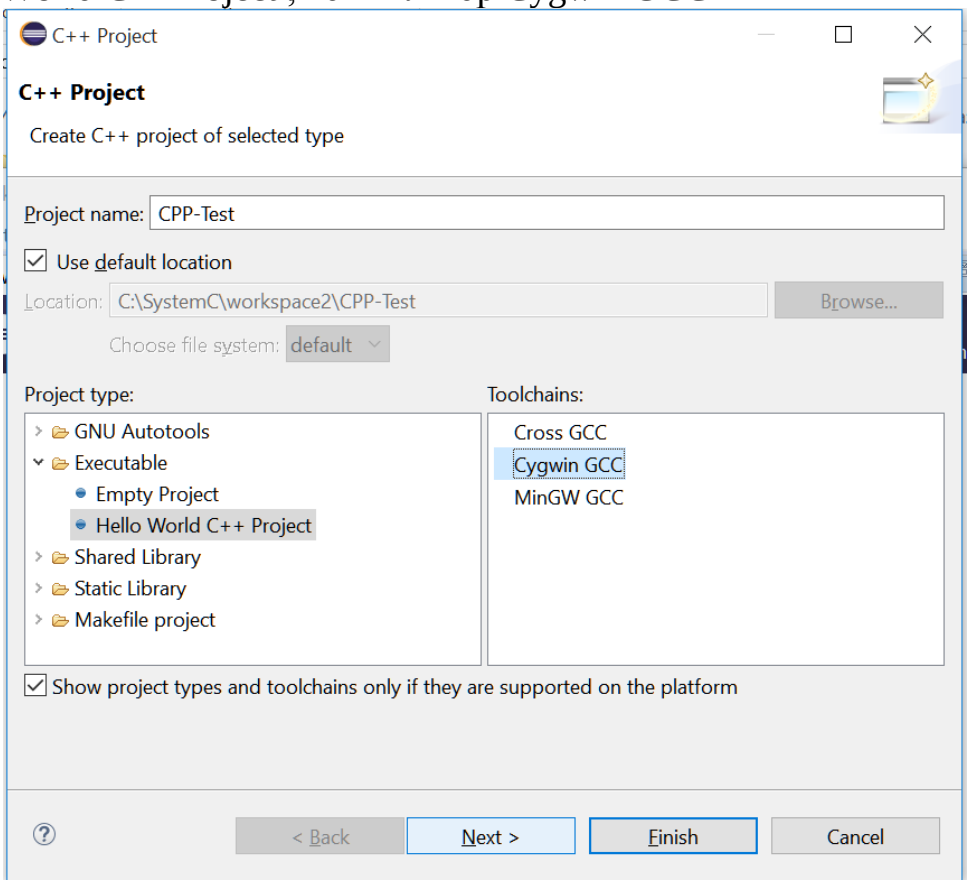


Рис. А.1

Основные свойства проекта отражены в окне (рис. А.2)

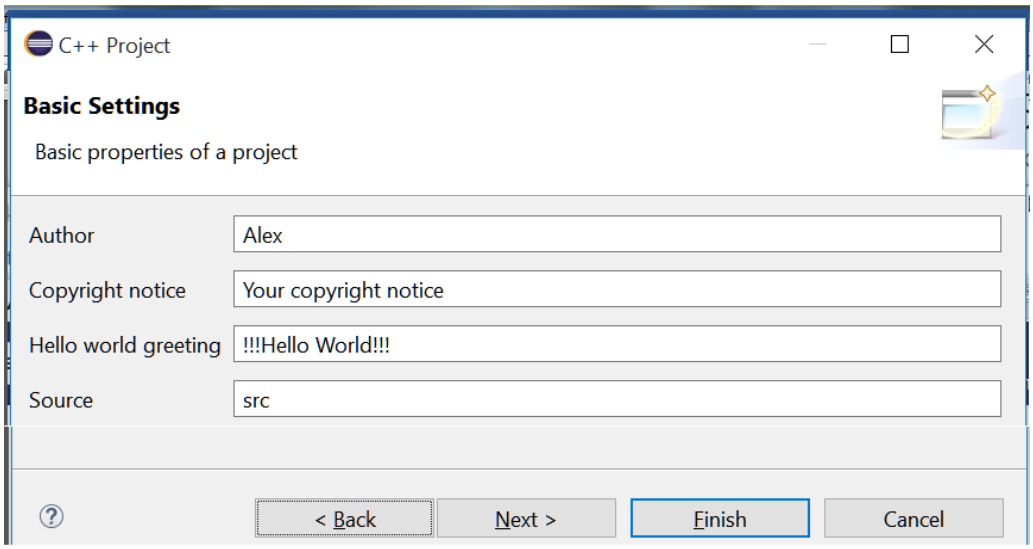


Рис. А.2

Устанавливаем конфигурации Debug и Release.

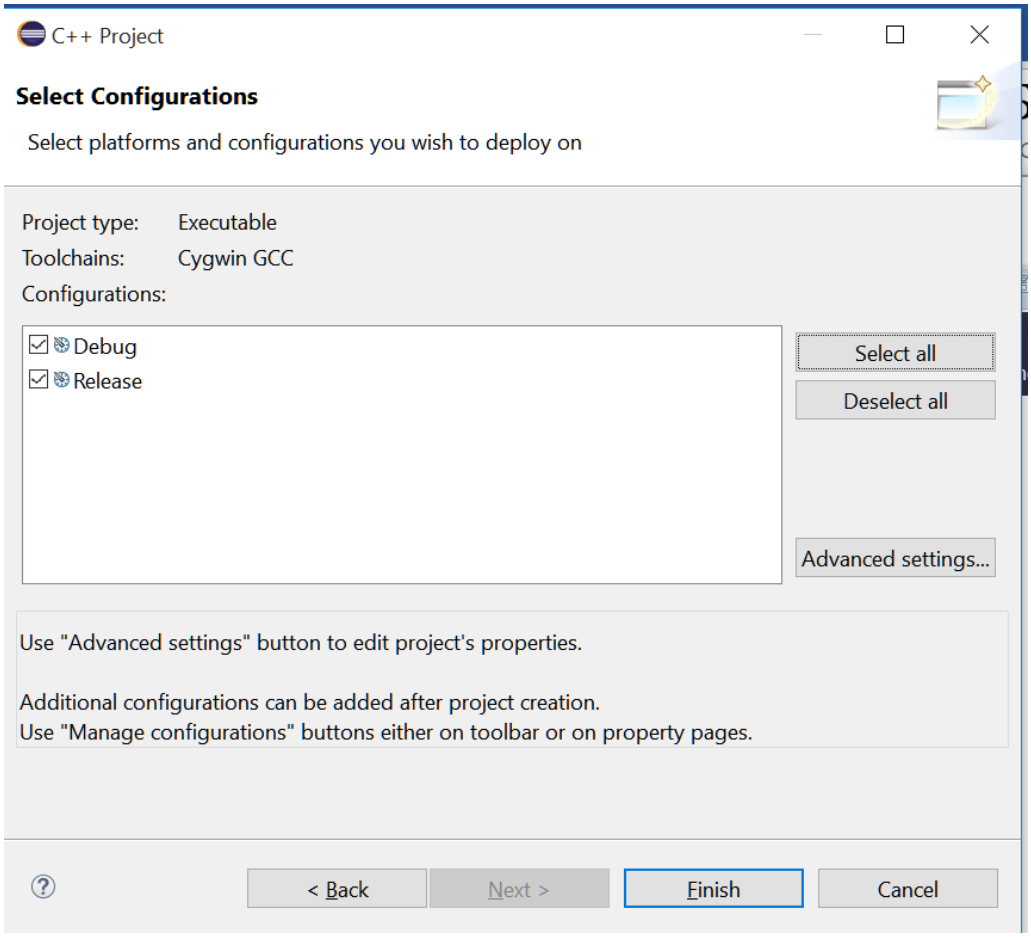


Рис. А.3

Сначала проверим программу «Hello world !!!!», скопировав текст в папку `src`.

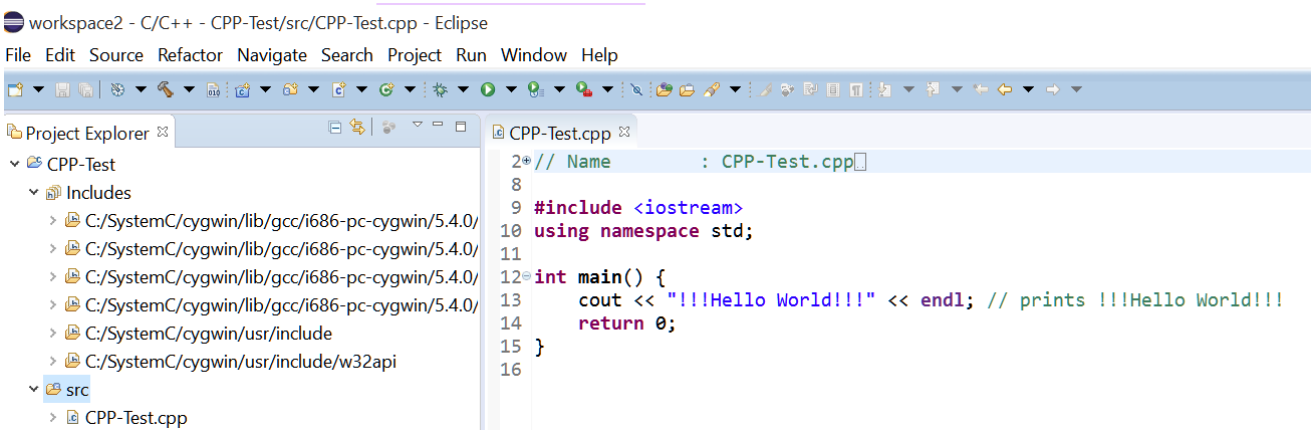


Рис. А.4

Выполняем компиляцию. Устанавливаем конфигурации Run (рис. А.5).

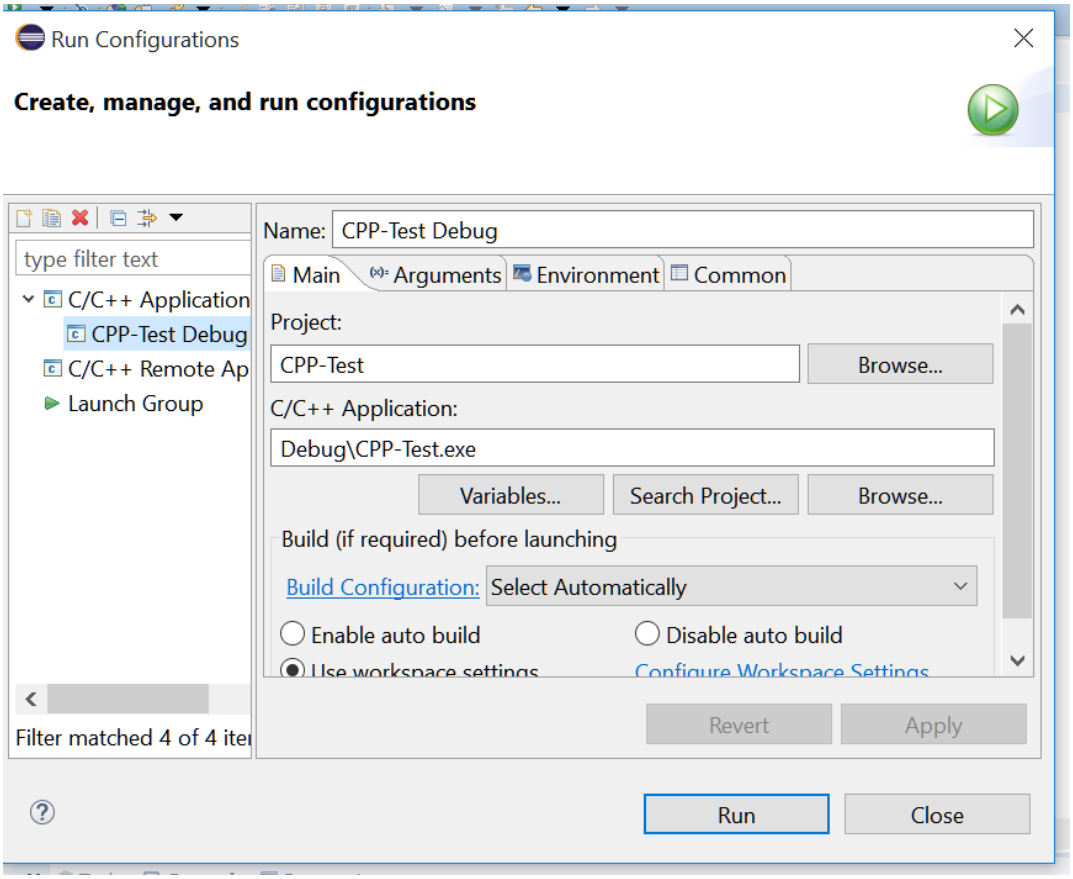


Рис. А.5

Выполняем решение и в консоли получаем (рис. А.6):

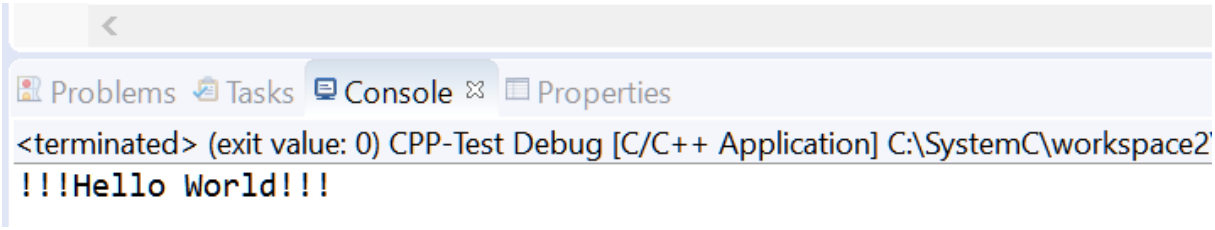


Рис. А.6

Запишем текст нашей программы из примера А.1. Чтобы ускорить выполнение этого, можно поступить так:

1. Копируем текст примера А.1 из этой книги, вставляем в Notepad++ и сохраняем как файл с расширением C++ source file (рис. А.7).

```
#include <iostream>
#include <cstdlib> // для system
using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
    system("pause"); // Только для тех, у кого MS Visual Studio
    return 0;
}
```

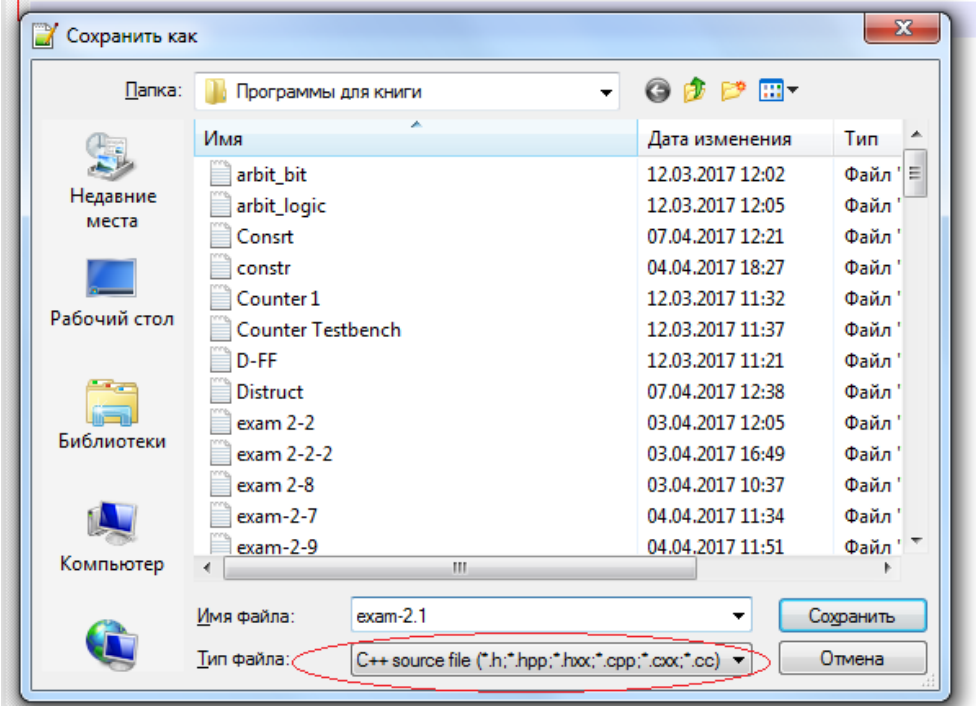


Рис. А.7

2. Вставляем файл в папку `src` проекта CPP-Test и переименовываем в исполняемый файл exam 2-1.cpp (рис. А.8).

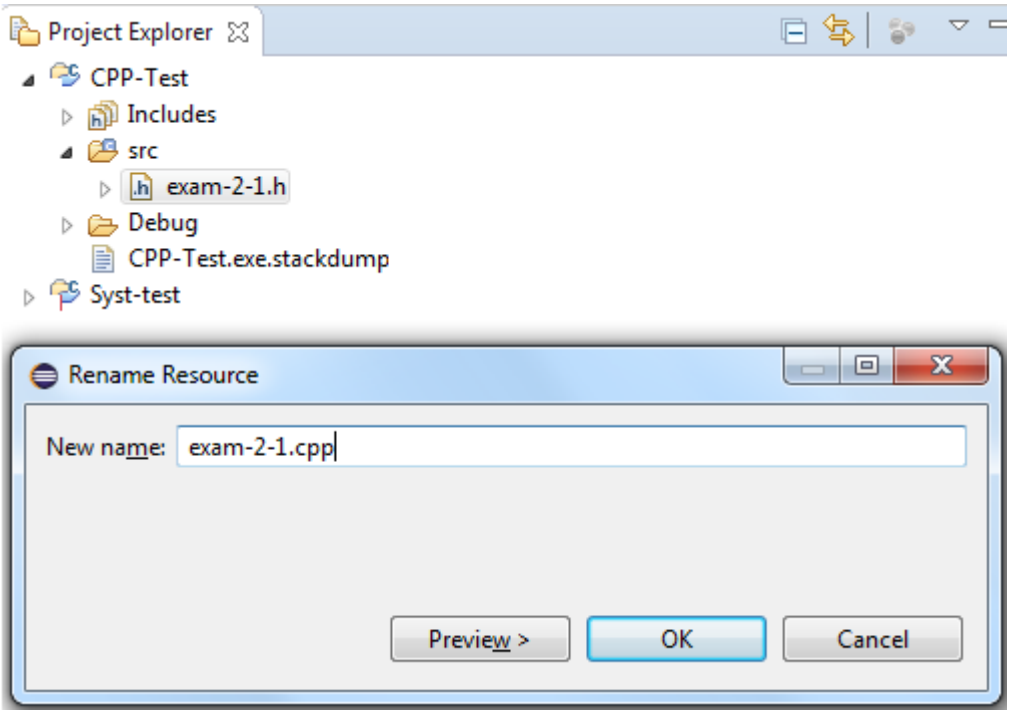


Рис. А.8

3. Открываем файл, выполняем компиляцию и решение (рис. А.9).

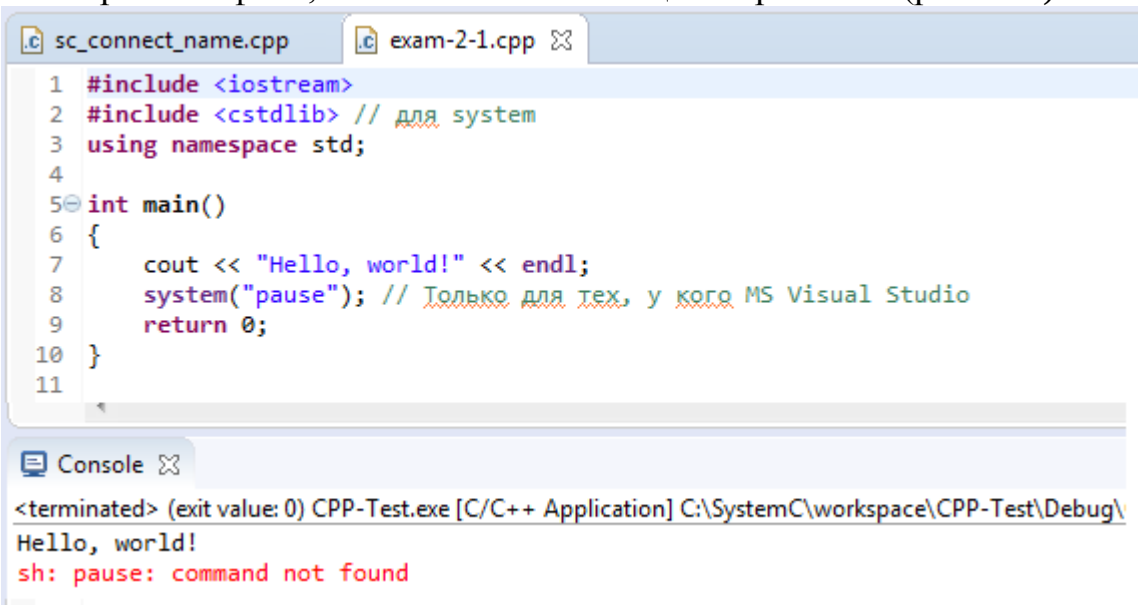


Рис. А.9

Аналогично выполним загрузку, компиляцию и решение примера А.2.

main.cpp

```
1        #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main()
7  {
8  int day_now, month_now, year_now;
9  cout << "Введите текущую дату и нажмите ENTER.\n";
10 cin >> day_now;
11 cout << "Введите месяц и нажмите ENTER.\n";
12 cin >> month_now;
13 cout << "Введите год и нажмите ENTER.\n";
14 cin >> year_now;
15 cout << day_now <<"."<< month_now <<"."<< year_now<< "\n"<<endl;
16
17 return 0;
18 }
```

Console

```
<terminated> (exit value: 0) Syst-test.exe [C/C++ Application] C:\SystemC\workspace\S
Введите текущую дату и нажмите ENTER.
29
Введите месяц и нажмите ENTER.
01
Введите год и нажмите ENTER.
2017
29.1.2017
```

Рис. А.10

А.2. Переменные C++

Переменная — это «ячейка» оперативной памяти компьютера, в которой может храниться какая-либо информация.

В программировании переменная, как и в математике, может иметь название, состоящее из одной латинской буквы, но также может состоять из нескольких символов, целого слова или нескольких слов.

А.2.1. Типы данных

В языке C++ *все переменные* имеют определенный тип данных. Например, переменная, имеющая целочисленный тип не может содержать ничего кроме целых чисел, а переменная с плавающей точкой — только дробные числа.

Тип данных присваивается переменной при ее объявлении или инициализации. Ниже приведены основные типы данных языка C++, которые нам понадобятся.

Основные типы данных в C++

int — целочисленный тип данных.

float — тип данных с плавающей запятой.
double — тип данных с плавающей запятой двойной точности.
char — символьный тип данных.
bool — логический тип данных.

Объявление переменной

Объявление переменной в C++ происходит таким образом: сначала указывается тип данных для этой переменной, а затем название этой переменной.

Пример объявления переменных

```
int a; // объявление переменной a целого типа.
float b; // объявление переменной b типа данных с плавающей запятой.
double c = 14.2; // инициализация переменной типа double.
char d = 's'; // инициализация переменной типа char.
bool k = true; // инициализация логической переменной k.
```

Заметьте, что в C++ **оператор присваивания** (=) — не является знаком равенства и не может использоваться для сравнения значений. Оператор равенства записывается как «двойное равно» (==).

Присваивание используется для сохранения определенного значения в переменной. Например, запись вида a = 10 задает переменной a значение числа 10.

Таблица А.1

Таблица типов данных

Тип	*Типичный размер в битах*	*Минимально допустимый диапазон значений*
char	8	от -127 до 127
unsigned char	8	от 0 до 255
signed char	8	от -127 до 127
int	16 или 32	от -32767 до 32767
unsigned int	16 или 32	от 0 до 65535
signed int	16 или 32	то же, что int
short int	16	от -32767 до 32767
unsigned short int	16	от 0 до 65535
signed short int	16	то же, что short int

long int	32	от -2 147 483 647 до 2 147 483 647
long long int	64	от $-(2^{63}-1)$ до $(2^{63}-1)$, добавлен стандартом C99
signed long int	32	то же, что long int
unsigned long int	32	от 0 до 4 294 967 295
unsigned long long int	64	от 0 до $(2^{64}-1)$, добавлен в C99
float	32	от 1E-37 до 1E+37, с точностью не менее 6 значащих десятичных цифр
double	64	от 1E-37 до 1E+37, с точностью не менее 10 значащих десятичных цифр
long double	80	от 1E-37 до 1E+37, с точностью не менее 10 значащих десятичных цифр

A.3. Операторы C++

Основные операторы языка C++ представлены в таблице A.2.

Таблица A.2

Унарные операции	
&	Операция взятия адреса
*	Операция обращения по ссылке
+	Унарный плюс
-	Унарный минус
~	Поразрядное дополнение (дополнение до единицы)
!	Логическое отрицание

++	Префикс: пред-инкремент, постфикс: пост-инкремент
- -	Префикс: пред-декремент, постфикс: пост-декремент
Бинарные операции	
+	Бинарный плюс (сложение)
-	Бинарный минус (вычитание)
*	Умножение
/	Деление
%	Остаток от деления
<<	Сдвиг влево, а также помещение в поток
>>	Сдвиг вправо, в также извлечение из потока
&	Поразрядное И
|	Поразрядное включающее ИЛИ
&&	Логическое И
||	Логическое ИЛИ
=	Присваивание
*=	Присвоить произведение
/=	Присвоить частное
%=	Присвоить остаток
+=	Присвоить сумму
-=	Присвоить разность
<<=	Присвоить сдвиг влево
>>=	Присвоить сдвиг вправо
&=	Присвоить поразрядное И
^=	Присвоить поразрядное исключающее ИЛИ

|=	Присвоить поразрядное ИЛИ
<	Меньше
>	Больше
<=	Меньше или равно
>=	Больше или равно
= =	Равно
!=	Не равно

В C++ операторы присваивания часто записывают в краткой форме.

Таблица А.3

Формы оператора присваивания

Обычная запись	Более кратко на C++
A=A*B	A*=B
A=A/(B+4)	A/=B+4
A=A%B	A%=B
A=A+B	A+=B
A=A-B	A-=B

А.4. Конструкции ветвления

Конструкции ветвления такие же как в С.

Условные операторы

В языке Си существуют два условных оператора: `if` и `switch`.
Общая форма оператора `if` следующая:

```
if (выражение) оператор;
else оператор;
```

Здесь оператор может быть только одним оператором, блоком операторов или отсутствовать (пустой оператор). Фраза `else` может вообще отсутствовать.

Если *выражение* истинно, то выполняется оператор или блок операторов, следующий за `if`. В противном случае выполняется оператор (или блок операторов), следующий за `else` (если эта фраза присутствует).

Условное выражение, входящее в `if`, должно иметь скалярный результат. Это значит, что результатом должно быть целое число, символ, указатель или число с плавающей точкой, но им не может быть массив или структура.

Пример А.3

```
#include <iostream>
using namespace std;

int main()
{
    setlocale(0, "");
    double num;

    cout << "Введите произвольное число: ";
    cin >> num;

   if (num < 10) { // Если введенное число меньше
10.
        cout << "Это число меньше 10." << endl;
    } else { // иначе
        cout << "Это число больше либо равно 10."
<< endl;
    }
    return 0;
}
```

Вложенные условные операторы if

Оператор `if` является вложенным, если он вложен, т.е. находится внутри другого оператора `if` или `else`. Во вложенном условном операторе фраза `else` всегда ассоциирована с ближайшим `if` в том же блоке, если этот if не ассоциирован с другой фразой else.

Лестница `if-else-if`

В программах часто используется конструкция, которую называют *лестницей* `if-else-if`. Общая форма лестницы имеет вид

```
if (выражение) оператор;
else
```

```
if (выражение) оператор;
else
  if (выражение) оператор;
  .
  .
  .
  else оператор;
```

Работает эта конструкция следующим образом. Условные выражения операторов `if` вычисляются сверху вниз. После выполнения некоторого условия, т.е. когда встретится выражение, принимающее значение ИСТИНА, выполняется ассоциированный с этим выражением оператор, а оставшаяся часть лестницы пропускается. Если все условия ложны, то выполняется оператор в последней фразе `else`, а если последняя фраза `else` отсутствует, то в этом случае не выполняется ни один оператор.

Пример А.4

```
#include <iostream>
using namespace std;

int main()
{
    setlocale(0, "");
    double num;

    cout << "Введите произвольное число: ";
    cin >> num;

    if (num < 10) // Если введенное число меньше
10.
        cout << "Это число меньше 10." << endl;
    else if (num == 10)
        cout << "Это число равно 10." << endl;
    else  // иначе
        cout << "Это число больше 10." << endl;

    return 0;
}
```

Оператор выбора – `switch`

Оператор выбора `switch` предназначен для выбора ветви вычислительного процесса исходя из значения управляющего выражения. При этом значение управляющего выражения сравнивается со значениями

в списке целых или символьных констант. Если будет найдено совпадение, то выполнится ассоциированный с совпавшей константой оператор. Общая форма оператора switch следующая:

```
switch (выражение) {
  case постоянная1:
    последовательность операторов
    break;
  case постоянная2:
    последовательность операторов
    break;
  case постоянная3:
    последовательность операторов
    break;
  default:
    последовательность операторов;
}
```

Значение *выражения* оператора `switch` должно быть таким, чтобы его можно было выразить целым числом. Это означает, что в управляющем выражении можно использовать переменные целого или символьного типа, но только не с плавающей точкой. Значение управляющего *выражения* по очереди сравнивается с *постоянными* в операторах `case.` Если значение управляющего *выражения* совпадет с какой-то из *постоянных*, управление передается на соответствующую метку `case` и выполняется *последовательность операторов* до оператора `break.` Если оператор `break` отсутствует, выполнение последовательности операторов продолжается до тех пор, пока не встретится `break` (в другой метке) или не кончится тело оператора `switch` (т.е. блок, следующий за `switch`). Оператор `default` выполняется в том случае, когда значение управляющего выражения не совпало ни с одной постоянной. Оператор `default` также может отсутствовать. В этом случае при отсутствии совпадений не выполняется ни один оператор.

Операторы цикла

В языке C++, как и в других языках программирования, операторы цикла служат для многократного выполнения последовательности операторов до тех пор, пока выполняется некоторое условие. Условие может быть установленным заранее (как в операторе `for`) или меняться при выполнении тела цикла (как в `while` или `do-while`).

Цикл `for`

Общая форма оператора for следующая:

```
for (инициализация; условие; приращение)
оператор;
```

Цикл for может иметь большое количество вариаций. В наиболее общем виде принцип его работы следующий. *Инициализация* — это присваивание начального значения переменной, которая называется параметром цикла. *Условие* представляет собой условное выражение, определяющее, следует ли выполнять оператор цикла (часто его называют телом цикла) в очередной раз. Оператор *приращение* осуществляет изменение параметра цикла при каждой итерации. Эти три оператора (они называются также секциями оператора for) обязательно разделяются точкой с запятой. Цикл for выполняется, если выражение *условие* принимает значение ИСТИНА. Если оно хотя бы один раз примет значение ЛОЖЬ, то программа выходит из цикла и выполняется оператор, следующий за телом цикла for.

В следующем примере в цикле for считается сумма чисел от 1 до 1000 и выводится результат.

Пример А.5

```
#include <iostream>
using namespace std;

int main()
{
    int i; // счетчик цикла
    int sum = 0; // сумма чисел от 1 до 1000.
    setlocale(0, "");
    for (i = 1; i <= 1000; i++) // задаем
начальное //значение 1, конечное 1000 и задаем шаг
цикла - 1.
    {
        sum = sum + i;
    }
    cout << "Сумма чисел от 1 до 1000 = " << sum
<< endl;
    return 0;
}
```

Бесконечный цикл

Для создания бесконечного цикла можно использовать любой оператор цикла, но чаще всего для этого выбирают оператор for. Так как

в операторе `for` может отсутствовать любая секция, бесконечный цикл проще всего сделать, оставив пустыми все секции. Это хорошо показано в следующем примере:

```
    for( ; ; ) printf("Этот цикл крутится
бесконечно.\n");
```

Если условие цикла `for` отсутствует, то предполагается, что его значение — ИСТИНА. В оператор `for` можно добавить выражения инициализации и приращения, хотя обычно для создания бесконечного цикла используют конструкцию

```
for( ; ; ).
```

Цикл `while`

Обшая форма цикла `while` имеет следующий вид:

```
while (условие) оператор;
```

Здесь *оператор* (тело цикла) может быть пустым оператором, единственным оператором или блоком. *Условие* (управляющее выражение) может быть любым допустимым в языке выражением. *Условие* считается истинным, если значение выражения не равно нулю, а *оператор* выполняется, если условие принимает значение ИСТИНА. Если условие принимает значение ЛОЖЬ, программа выходит из цикла и выполняется следующий за циклом оператор.

Пример А.6

```
#include <iostream>
using namespace std;

int main()
{
    setlocale(0, "");
    int i = 0; // инициализируем счетчик цикла.
    int sum = 0; // инициализируем счетчик суммы.
    while (i < 1000)
    {
        i++;
        sum += i;
    }
    cout << "Сумма чисел от 1 до 1000 = " << sum
<< endl;
    return 0;
}
```

Цикл `do-while`

В отличие от циклов `for` и `while`, которые проверяют свое условие перед итерацией, `do-while` делает это после нее. Поэтому цикл `do-while` всегда выполняется как минимум один раз. Общая форма цикла `do-while` следующая:

```
do {
оператор;
} while (условие);
```

Если оператор не является блоком, фигурные скобки не обязательны, но их почти всегда ставят, чтобы оператор достаточно наглядно отделялся от условия. Итерации оператора `do-while` выполняются, пока условие не примет значение ЛОЖЬ.

Пример А.7.

Программа с циклом `do-while`

```
#include <iostream>
using namespace std;

int main ()
{
    setlocale(0, "");
    int i = 0; // инициализируем счетчик цикла.
    int sum = 0; // инициализируем счетчик суммы.
    do {// выполняем цикл.
        i++;
        sum += i;
    } while (i < 1000); // пока выполняется
условие.
    cout << "Сумма чисел от 1 до 1000 = " << sum
<< endl;
    return 0;
}
```

Выполните рещение в Eclipse. Результат решения показан на рис. А.11.

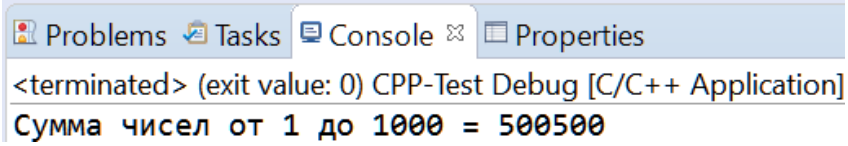


Рис. А.11

А.5. Массивы в C++

Массивы могут быть определенны с использованием любого типа данных. Общая форма определения массива имеет следующий вид.

```
тип Имя_массива[размер];
```

Здесь *тип* определяет тип данных для каждого элемента в массиве, а *размер* указывает количество элементов в массиве. Например, чтобы определить массив x из 100 элементов, запишите следующее.

```
int x[100];
```

Эта запись создаёт массив, содержащий 100 элементов, причём номер первого элемента — 0, а последнего — 99.

Многомерные массивы объявляются посредством размещения дополнительных измерений внутри дополнительных квадратных скобок. Например, чтобы определить массив 10×20, необходимо записать

```
int x[10][20];
```

Массивы можно инициализировать с помощью списка инициализаторов, заключённого в фигурные скобки, как, например, показано ниже.

```
int mass[3][2] = {{1, 2},
                  {3, 4},
                  {5, 6},
                 }
```

При объявлении массивов с неизменным количеством элементов можно не указывать размер только в самых левых скобках.

```
int mass[][3] = {1, 2, 3,
                 4, 5, 6,
                 7, 8, 9};
```

В современном стандарте C++ определен класс с функциями и свойствами (переменными) для организации работы со строками (в классическом языке C строк как таковых нет, есть лишь массивы символов `char`):

```
#include <string>
```

Для работы со строками также нужно подключить стандартный `namespace:`

```
using namespace std;
```

В противном случае придётся везде указывать описатель класса `std::string` вместо `string`.

Ниже приводится пример программы, работающей со `string` (в старых си-совместимых компиляторах не работает!).

Пример А.8

```
#include <iostream>
#include <string>
#include <malloc.h>
using namespace std;

int main () {
```

```
 string s = "Test";
 s.insert (1,"!");
 cout << s.c_str() << endl;
 string *s2 = new string("Hello");
 s2->erase(s2->end());
 cout << s2->c_str();
 cin.get(); return 0;
}
```

Массив (string) — это область памяти, где могут последовательно храниться несколько значений.

Возьмем группу студентов из десяти человек. У каждого из них есть фамилия. Создавать отдельную переменную для каждого студента — не рационально. Создадим массив, в котором будут храниться фамилии всех студентов.

```
string students[10] = {
    "Иванов", "Петров", "Сидоров",
    "Ахмедов", "Ерошкин", "Выхин",
    "Андеев", " Вин Дизель", " Картошкин", "
Чубайс"
};
```

Заметьте, что нумерация элементов массива в C++ начинается с нуля. Следовательно, фамилия первого студента находится в `students[0]`, а фамилия последнего — в `students[9]`.

В большинстве языков программирования нумерация элементов массива также начинается с нуля.

Вывод элементов массива через цикл

Пример А.9

```
#include <iostream>
#include <string>

int main()
{
     std::string students[10] = {
          "Иванов", "Петров", "Сидоров",
          "Ахмедов", "Ерошкин", "Выхин",
```

```
                "Андеев", "Вин Дизель", "Картошкин",
"Чубайс"
        };
        for (int i = 0; i < 10; i++) {
          std::cout << students[i] << std::endl;
      }

        return 0;
}
```

Выполним решение примера А.9 в Eclipse.

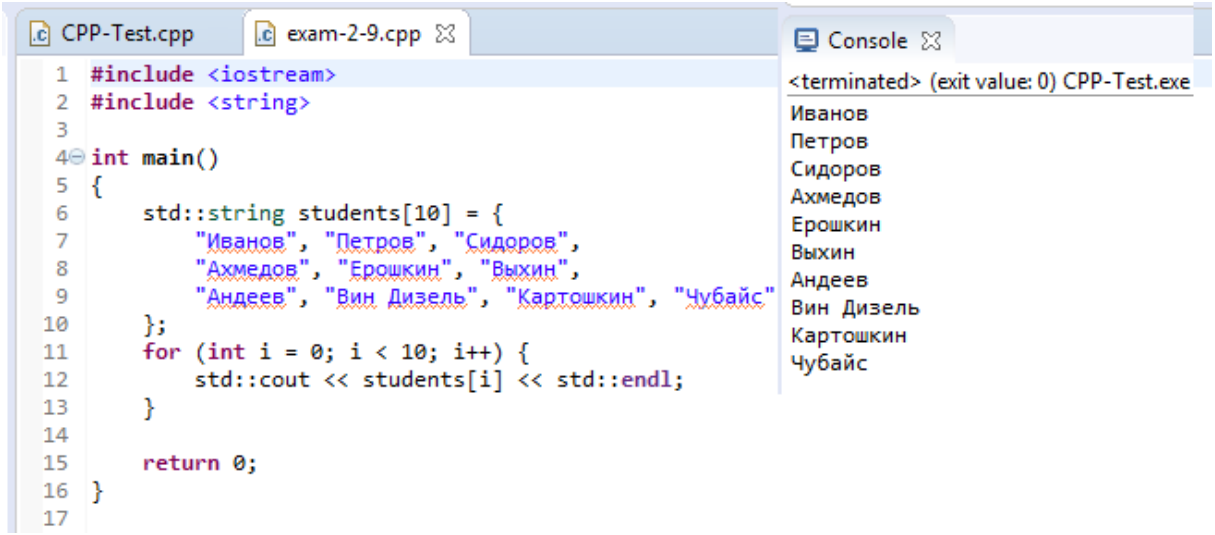


Рис. А.12

А.6. Функции в C++

Функции — один из самых важных компонентов языка C++.

Любая функция имеет тип, также, как и любая переменная.

Функция может возвращать значение, тип которого в большинстве случаев аналогичен типу самой функции.

Если функция не возвращает никакого значения, то она должна иметь тип `void` (такие функции иногда называют процедурами).

При объявлении функции, после ее типа должно находиться имя функции и две круглые скобки — открывающая и закрывающая, внутри которых могут находиться один или несколько аргументов функции, которых также может не быть вообще.

После списка аргументов функции ставится открывающая фигурная скобка, после которой находится само тело функции.

В конце тела функции обязательно ставится закрывающая фигурная скобка.

Пример построения функции

```
#include <iostream>
using namespace std;

void function_name ()
{
   cout << "Hello, world" << endl;
}

int main()
{
    function_name(); // Вызов функции
   return 0;
}
```

Перед вами тривиальная программа, **Hello, world**, только реализованная с использованием функций.

Если мы хотим вывести «Hello, world» где-то еще, нам просто нужно вызвать соответствующую функцию. В данном случае это делается так: `function_name();`. Вызов функции имеет вид имени функции с последующими круглыми скобками. Эти скобки могут быть пустыми, если функция не имеет аргументов. Если же аргументы в самой функции есть, их необходимо указать в круглых скобках.

Также существует такое понятие, как параметры функции по умолчанию. Такие параметры можно не указывать при вызове функции, т.к. они примут значение по умолчанию, указанные после знака присваивания данного параметра и в списке всех параметров функции.

В предыдущих примерах мы использовали функции типа `void`, которые не возвращают никакого значения. Оператор `return` используется для возвращения вычисляемого функцией значения.

Рассмотрим пример функции, возвращающей значение на примере проверки пароля.

Пример А.10

```
#include <iostream>
#include <string>

using namespace std;
```

```
string check_pass (string password)
{
    string valid_pass = "qwerty123";
    string error_message;
    if (password == valid_pass) {
        error_message = "Доступ разрешен.";
    } else {
        error_message = "Неверный пароль!";
    }
    return error_message;
}

int main()
{
    string user_pass;
    cout << "Введите пароль: ";
    getline (cin, user_pass);
    string error_msg = check_pass (user_pass);
    cout << error_msg << endl;
    return 0;
}
```

В данном случае функция `check_pass` имеет тип `string`, следовательно она будет возвращать только значение типа `string`, иными словами говоря строку. Давайте рассмотрим алгоритм работы этой программы.

Самой первой выполняется функция `main()`, которая должна присутствовать в каждой программе. Теперь мы объявляем переменную `user_pass` типа `string`, затем выводим пользователю сообщение «Введите пароль», который после ввода попадает в строку `user_pass.` А вот дальше начинает работать наша собственная функция `check_pass().`

В качестве аргумента этой функции передается строка, введенная пользователем.

Аргумент функции — это переменные или константы вызывающей функции, которые будет использовать вызываемая функция.

При объявлении функций создается **формальный параметр**, имя которого может отличаться от параметра, передаваемого при вызове этой функции. Но типы формальных параметров и передаваемых функции аргументов в большинстве случаев должны быть аналогичны.

После того, как произошел вызов функции `check_pass(),` начинает работать данная функция. Если функцию нигде не вызывать, то

этот код будет проигнорирован программой. Итак, мы передали в качестве аргумента строку, которую ввел пользователь.

Теперь эта строка в полном распоряжении функции. Следует понимать, что переменные и константы, объявленные в разных функциях независимы друг от друга, они даже могут иметь одинаковые имена. Их действие регулируется понятиями: видимости, локальные и глобальные переменные.

Теперь мы проверяем, правильный ли пароль ввел пользователь или нет. если пользователь ввел правильный пароль, присваиваем переменной `error_message` соответствующее значение, если нет, то сообщение об ошибке.

После этой проверки мы **возвращаем** переменную `error_message`. На этом работа нашей функции закончена. А теперь, в функции `main(),` то значение, которое возвратила наша функция мы присваиваем переменной `error_msg` и выводим это значение (строку) на экран терминала.

Рекомендуем выполнить решение в Eclipse с компилятором GCC.

A.7. Указатели в C++. Статические и динамические переменные

В большинстве языков, в том числе и C/C++, имеется понятие указателя. Указатель — это переменная, хранящая в себе адрес ячейки оперативной памяти, например **0x100.**

Мы можем обращаться, например, к массиву данных через указатель, который будет содержать адрес начала диапазона ячеек памяти, хранящих этот массив.

После того, как этот массив станет не нужен для выполнения остальной части программы, мы просто освободим память по адресу этого указателя, и она вновь станет доступна для других переменных.

Ниже приведен конкретный пример обращения к переменным через указатель и напрямую.

A.7.1. Пример использования статических переменных

Пример A.11

```
#include <iostream>
using namespace std;

int main()
{
    int a; // Объявление статической переменной
    int b = 5; /* Инициализация статической
переменной b*/
```

```
    a = 10;
    b = a + b;
    cout << "b is " << b << endl;
    return 0;
}
```

Результат решения

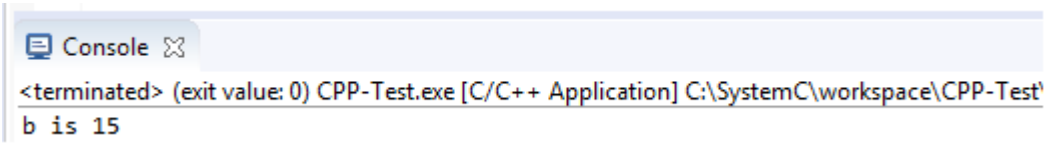


Рис. А.13

А.7.2. Пример использования динамических переменных

Пример А.12

```
#include <iostream>
using namespace std;

int main()
{
    int *a = new int; /* Объявление указателя для переменной типа int*/
    int *b = new int(5); /* Инициализация указателя*/

    *a = 10;
    *b = *a + *b;

    cout << "b is " << *b << endl;

    delete b;
    delete a;

    return 0;
}
```

Во втором примере мы оперируем динамическими переменными посредством указателей. Получаем тот же ответ в решении.

Рассмотрим общий синтаксис указателей в C++.

Выделение памяти осуществляется с помощью оператора `new` и имеет вид: `тип_данных *имя_указателя = new тип_данных;`, например `int *a = new int;`. После выполнения такой операции, в

оперативной памяти компьютера происходит выделение диапазона ячеек, необходимого для хранения переменной типа `int`.

Инициализация значения, находящегося по адресу указателя выполняется схожим образом, только в конце ставятся круглые скобки с нужным значением:

`тип данных *имя_указателя = new тип_данных(значение).`

В нашем примере это `int *b = new int(5).`

Для того, чтобы получить **адрес** в памяти, на который ссылается указатель, используется имя переменной-указателя с префиксом `&` перед ним .

Например, чтобы вывести на экран адрес ячейки памяти, на который ссылается указатель `b` во втором примере, мы пишем:

```
cout << "Address of b is " << &b << endl;
```

Чтобы изменить значение, находящееся по адресу, на который ссылается указатель, нужно также использовать звездочку, например, как во втором примере:

```
*b = *a + *b;.
```

Когда мы оперируем **данными**, то используем знак `*`

Когда мы оперируем **адресами**, то используем знак `&`

Для того, чтобы освободить память, выделенную оператором **new**, используется оператор `delete.`

Пример освобождения памяти

```
#include <iostream>
using namespace std;

int main()
{
   // Выделение памяти
   int *a = new int;
   int *b = new int;
   float *c = new float;

   // ... Любые действия программы

   // Освобождение выделенной памяти
   delete c;
   delete b;
   delete a;
```

```
    return 0;
}
```

A.8. Запуск из командной строки

При запуске программы из командной строки, ей можно передавать дополнительные параметры в текстовом виде.

В программе эти параметры из командной строки можно получить через аргументы функции `main` при использовании функции `main` в следующей форме:

```
int main(int argc, char* argv[]) { /* ... */ }
```

Первый аргумент содержит количество параметров командной строки. Второй аргумент — это массив строк, содержащий параметры командной строки. Т.е. первый аргумент указывает количество элементов массива во втором аргументе.

Первый элемент массива строк (`argv[0]`) всегда содержит строку, использованную для запуска программы (либо пустую строку). Следующие элементы (от `1 до argc - 1`) содержат параметры командной строки, если они есть. Элемент массива строк `argv[argc]` всегда должен содержать 0.

Пример А.13

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        // Выводим список аргументов в цикле
        cout << "Argument " << i << " : " <<
argv[i] << endl;
    }
    return 0;
}
```

A.9. Структуры в C++

Структура — это объединение различных переменных (даже с разными типами данных), которому можно присвоить имя. Например, можно объединить данные об объекте Дом: город (в котором дом находится), улица, количество квартир и т.д. в одной структуре. Можно

собрать в одну совокупность данные обо всем, что необходимо конкретному программисту.

Сначала, Вам необходимо ознакомиться с синтаксисом структур в языке C++. Рассмотрим простой пример, который поможет познакомиться со структурами и покажет, как с ними работать. Создадим структуру, создадим объект структуры, заполним значениями элементы структуры (данные об объекте) и выведем эти значения на экран.

CPP-Test.cpp | struct.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  struct building     //Создаем структуру!
5  {
6      char *owner;        //здесь будет храниться имя владельца
7      char *city;         //название города
8      int amountRooms;    //количество комнат
9      float price;        //цена
10 };
11
12 int main()
13 {
14     setlocale (LC_ALL, "rus");
15
16     building apartment1;   //это объект структуры с типом данных, именем структуры, building
17
18     apartment1.owner = "Денис"; //заполняем данные о владельце и т.д.
19     apartment1.city = "Симферополь";
20     apartment1.amountRooms = 5;
21     apartment1.price = 150000;
22
23     cout << "Владелец квартиры: " << apartment1.owner << endl;
24     cout << "Квартира находится в городе: " << apartment1.city << endl;
25     cout << "Количество комнат: " << apartment1.amountRooms << endl;
26     cout << "Стоимость: " << apartment1.price << " $" << endl;
27
28 return 0;
29 }
```

Console

```
<terminated> (exit value: 0) CPP-Test.exe [C/C++ Application] C:\SystemC\workspace\CPP-Test\Debug\CPP-Test.exe (05.04.17, 14:57)
Владелец квартиры: Денис
Квартира находится в городе: Симферополь
Количество комнат: 5
Стоимость: 150000 $
```

Рис. А.17

Комментарии по коду программы:

В **строках 4 — 10** мы создаем структуру. Чтобы ее объявить используем зарезервированное слово `struct` и даем ей любое имя. В нашем случае – `building`.

Далее открываем фигурную скобку `{`, перечисляем 4 элемента структуры через точку с запятой `;`, закрываем фигурную скобку `}` и в завершении ставим точку с запятой `;`. Это будет шаблоном (формой) структуры.

В **строке 16** объявляем объект структуры. Как и для обычных переменных, необходимо объявить тип данных. В этом качестве выступит имя нашей созданной структуры — `building`.

Заполняем данными (инициализируем) элементы структуры: имя объекта далее оператор точка «.» и имя элемента структуры. Например: `apartment1.owner`. Таким образом, в **строках 18-21** присваиваем данные элементам структуры.

Чтобы обратиться к элементам структуры так же, как и при инициализации, используем точку «.» и имя элемента структуры. В строках 23 — 26 выводим заполненные элементы структуры на экран.

А.10. Классы в С++

Практически любой материальный предмет можно представить в виде совокупности объектов, из которых он состоит. Допустим, что нам нужно написать программу для учета успеваемости студентов. Можно представить группу студентов, как класс языка С++. Назовем его `Students`.

```
class Students {
    // Имя студента
    std::string name;
    // Фамилия
    std::string last_name;
    // Пять промежуточных оценок студента
    int scores[5];
    // Итоговая оценка за семестр
    float average_ball;
};
```

Основные понятия

Классы в программировании состоят из свойств и методов. Свойства — это любые данные, которыми можно характеризовать объект класса.

Классы являются основой С++. Перед тем, как создать объект С++, необходимо определить его общую форму, используя ключевое слово `class`. Класс определяет новый тип данных, который соединяет в себе код и данные. По синтаксису класс аналогичен структуре. Однако класс также может включать функции, а не только данные. В качестве примера следующий класс определяет тип, который мы называем `queue` (очередь).

```
// создание класса очередь
Class queue {
int q[100];
```

```
int sloc, rloc;
public: void init ();
void qput(int i);
int qget();
};
```

Класс может содержать публичные и частные части. По умолчанию все члены класса являются частными. Например, переменные `q, sloc` и `rloc` являются частными, то есть функции, не являющиеся членами класса, не имеют к ним доступа. Именно так достигается инкапсуляция: доступ к определенным частям данных может быть строго контролируемым. Хотя это и не показано в данном примере, можно также определить частные функции, которые могут вызываться только членами данного класса.

Для того, чтобы сделать части класса доступными из других частей программы, они должны быть объявлены с использованием ключевого слова `public`, которое и служит спецификатором доступа. Все переменные или функции, определенные после ключевого слова `public`, являются публичными, т.е. доступны для всех других функций в программе. В общем случае доступ к объекту из остальной части программы осуществляется с помощью функций со спецификатором доступа `public`. Хотя можно иметь переменные с публичным доступом, лучше ограничить их использование или вовсе исключить из употребления. Вместо этого следует сделать все данные частными и контролировать доступ к ним с помощью функций, имеющих спецификатор доступа `public`. Таким образом публичные функции обеспечивают интерфейс к частным данным класса. Это помогает реализовать инкапсуляцию. Обратим внимание также, что после ключевого слова `public` следует двоеточие.

Функции `init(), qput() и qget()` называются функциями-членами, потому что они являются частью класса `queue`. Переменные `sloc, rloc` и `q` называются переменными-членами (или членами данных). Только функции-члены имеют доступ к частным членам класса, в котором они объявлены. Таким образом, только `init(), qput()` и `qget()` имеют доступ к `sloc`, `rloc` и `q`.

Как только определен класс, можно создать объект этого типа, используя имя класса. Фактически имя класса становится спецификатором нового типа данных. Например, следующий код создает объект с именем `intqueue` типа `queue`:

```
queue intqueue;
```

Также можно создать объекты при определении класса, помещая имена переменных после закрывающей фигурной скобки, в точности так

же, как это имеет место для структуры. Общий вид объявления класса следующий:

class имя_класса {
private данные и функции
public:
публичные данные и функции
} список объектов;

Список объектов может быть пустым.

Например, ниже показан один из способов написания функции `qput()`:

```
void queue::qput(int i)
{
if (sloc==99) {
cout << "Queue is full.\n";
return;
}
sIoc++;
q[sloc] = i;
}
```

Последовательность символов `::` называется оператором области видимости (`scope resolution operator`). Он показывает принадлежность функции классу. В данном случае он говорит компилятору, что данная версия функции `qput()` принадлежит классу queue. Другими словами, функция `qput()` находится в области видимости класса `queue`. В C++ несколько различных классов могут использовать одинаковые имена функций. Компилятор знает, какая из них принадлежит какому классу благодаря оператору области видимости и имени класса.

Если вы рассматриваете часть программы, которая не входит в состав класса, то для вызова функции-члена класса необходимо использовать имя объекта и оператор «точка». Например, следующий фрагмент иллюстрирует вызов функции `init()` для объекта `a`:

```
queue a, b;
a.init();
```

Очень важно ясно себе представлять, что `a` и `b` являются двумя различными объектами. Это означает, что инициализация `a` не означает инициализацию `b`. Единственной связью между объектами `a` и `b` служит то, что они являются объектами одного и того же типа. Более того, копии переменных `sloc, rloc` и `q` объекта `a` совершенно независимы от соответствующих копий переменных объекта `b`.

Использование имени класса и оператора «точка» необходимо только при вызове функции-члена извне класса. Внутри класса одна

функция-член может вызывать другую функцию-член непосредственно без использования оператора «точка». Аналогично функция-член может обращаться непосредственно к переменной-члену без использования оператора «точка». Представленная ниже программа демонстрирует все части класса queue:

Вернемся к программе, где объектом класса является студент, а его свойствами — имя, фамилия, оценки и средний балл.

У каждого студента есть имя — name и фамилия last_name . Также, у него есть промежуточные оценки за весь семестр. Эти оценки мы будем записывать в целочисленный массив из пяти элементов. После того, как все пять оценок будут проставлены, определим средний балл успеваемости студента за весь семестр — свойство average_ball.

Методы — это функции, которые могут выполнять какие-либо действия над данными (свойствами) класса. Добавим в наш класс функцию calculate_average_ball(), которая будет определять средний балл успеваемости ученика.

Методы класса – это его функции.

Свойства класса – это его переменные.

```
class Students {
    public:
        // Функция, считающая средний балл
        void calculate_average_ball()
        {
            int sum = 0; // Сумма всех оценок
            for (int i = 0; i < 5; ++i) {
                sum += scores[i];
            }
            // считаем среднее арифметическое
            average_ball = sum / 5.0;
        }

        // Имя студента
        std::string name;
        // Фамилия
        std::string last_name;
        // Пять промежуточных оценок студента
        int scores[5];

    private:
        // Итоговая оценка за семестр
        float average_ball;
```

```
};
```

Функция `calculate_average_ball()` просто делит сумму всех промежуточных оценок на их количество.

A.10.1. Модификаторы доступа public и private

Все свойства и методы классов имеют права доступа. По умолчанию, все содержимое класса является доступным для чтения и записи только для него самого. Для того, чтобы разрешить доступ к данным класса извне, используют модификатор доступа `public.` Все функции и переменные, которые находятся после модификатора `public,` становятся доступными из всех частей программы.

Закрытые данные класса размещаются после модификатора доступа `private.` Если отсутствует модификатор `public,` то все функции и переменные, по умолчанию являются закрытыми (как в первом примере).

Обычно, приватными делают все свойства класса, а публичными — его методы. Все действия с закрытыми свойствами класса реализуются через его методы. Рассмотрим следующий код.

```
class Students {
    public:
        // Установка среднего балла
        void set_average_ball(float ball)
        {
            average_ball = ball;
        }
        // Получение среднего балла
        float get_average_ball()
        {
            return average_ball;
        }
        std::string name;
        std::string last_name;
        int scores[5];

    private:
        float average_ball;
};
```

Мы не можем напрямую обращаться к закрытым данными класса. Работать с этими данными можно только посредством методов этого класса. В примере выше, мы используем функцию `get_average_ball()` для получения средней оценки студента, и `set_average_ball()` для выставления этой оценки.

Функция `set_average_ball()` принимает средний балл в качестве параметра и присваивает его значение закрытой переменной `average_ball.` Функция `get_average_ball()` просто возвращает значение этой переменной.

A.11. Создание объекта через указатель

При создании объекта, лучше не копировать память для него, а выделять ее с помощью указателя. И освобождать ее после того, как мы закончили работу с объектом. Реализуем это в нашей программе, немного изменив содержимое файла `main.cpp.`

```
    /* main.cpp */
    #include <iostream>
    #include "students.h"

    int main()
    {
        // Выделение памяти для объекта Students
        Students *student = new Students;

        std::string name;
        std::string last_name;

        // Ввод имени с клавиатуры
        std::cout << "Name: ";
        getline(std::cin, name);
    //
    //
    //

             << student->get_last_name() << " is "
             << student->get_average_ball() <<
std::endl;
        // Удаление объекта student из памяти
        delete student;
        return 0;
    }
```

При создании статического объекта, для доступа к его методам и свойствам, используют операция прямого обращения — « . » (символ точки). Если же память для объекта выделяется посредством указателя, то для доступа к его методам и свойствам используется оператор косвенного обращения — « -> » .

A.13. Конструктор и деструктор класса

Конструктор— специальная функция, которая выполняет начальную инициализацию элементов данных, причём имя конструктора обязательно должно совпадать с именем класса. Важным отличием конструктора от остальных функций является то, что он не возвращает значений вообще никаких, в том числе и `void`. В любом классе должен быть конструктор, даже если явным образом конструктор не объявлен, то компилятор предоставляет конструктор по умолчанию, без параметров.

Деструктор класса вызывается при уничтожении объекта. Имя деструктора аналогично имени конструктора, только в начале ставится знак тильды ~ . Деструктор не имеет входных параметров.

Важно соблюдать такие правила:

1. Конструктор и деструктор, мы всегда объявляем в разделе `public`;
2. При объявлении конструктора, тип данных возвращаемого значения не указывается, в том числе — `void`;
3. У деструктора также нет типа данных для возвращаемого значения, к тому же деструктору нельзя передавать никаких параметров;
4. Имя класса и конструктора должно быть идентично;
5. Имя деструктора идентично имени конструктора, но должно иметь приставку ~ ;
6. В классе допустимо создавать несколько конструкторов, если это необходимо. Имена будут одинаковыми. Компилятор будет их различать по передаваемым параметрам (как при перегрузке функций). Если мы не передаем в конструктор параметры, он считается конструктором по умолчанию;
7. Обратите внимание на то, что в классе может быть объявлен только один деструктор.

Пример A.15

```
# include <iostream>
using namespace std;

class AB //класс
{
```

```
        private:
        int a;
        int b;
        public:
        AB()    //это конструктор:  1) у конструктора
нет типа возвращаемого значения! в том числе void!!!
        //   2) имя должно быть таким как и у класса
(в нашем случае AB)
        {
            a = 0;//присвоим начальные значения
переменным
            b = 0;
            cout << "Работа конструктора при создании
нового объекта: " << endl;//и здесь же их отобразим
на экран
            cout << "a = " << a << endl;
            cout << "b = " << b << endl << endl;
        }

        void setAB() // с помощью этого метода
изменим начальные значения заданные конструктором
        {
            cout << "Введите целое число a: ";
            cin >> a;
            cout << "Введите целое число b: ";
            cin >> b;
        }

        void getAB() //выведем на экран измененные
значения
        {
            cout << "a = " << a << endl;
            cout << "b = " << b << endl << endl;
        }
    };

    int main()
    {
        setlocale(LC_ALL, "rus");

        AB obj1;     //конструктор сработает на
данном этапе (во время создания объекта класса)
```

```
        obj1.setAB();    //присвоим новые значения
переменным
        obj1.getAB();    //и выведем их на экран

        AB obj2;      //конструктор сработает на
данном этапе (во время создания 2-го объекта класса)
    return 0;
    }
```

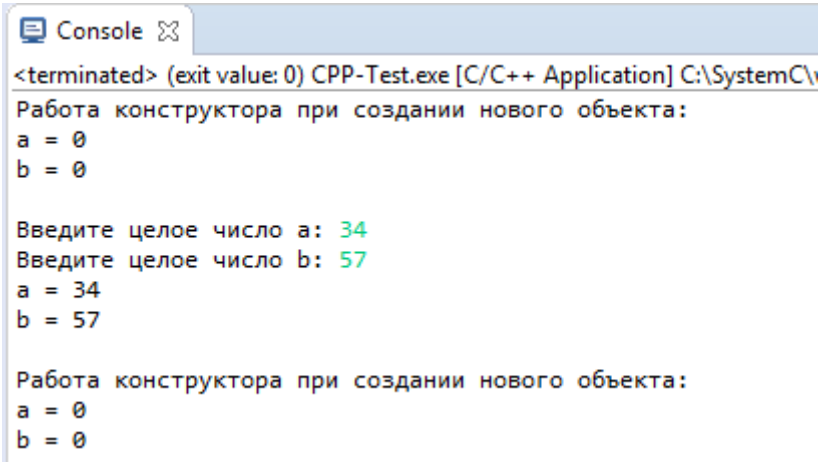


Рис. А.18. Применение конструктора

Как видно из результата работы программы, конструктор срабатывает сразу, при создании объектов класса, поэтому, явно вызывать конструктор не нужно.

Как и обычным функциям, мы можем передавать конструктору параметры. Через параметры, конструктору можно передавать любые данные, которые будут необходимы при инициализации объектов класса.

Рассмотрим еще один пример, это все та же программа, только в код внесены некоторые изменения. Тут же покажем принцип работы деструктора.

Пример А.16

```
# include <iostream>
using namespace std;

class AB //класс
{
    private:
    int a;
    int b;

    public:
```

```
        AB(int A, int B) /*эти параметры мы передадим
при создании объекта в main*/
        {
            a = A;/*присвоим нашим элементам класса
значения параметров*/
            b = B;
            cout << "Тут сработал конструктор,
который принимает параметры: " << endl;/*и здесь же
их отобразим на экран*/
            cout << "a = " << a << endl;
            cout << "b = " << b << endl << endl;
        }

        void setAB()
        {
            cout << "Введите целое число a: ";
            cin >> a;
            cout << "Введите целое число b: ";
            cin >> b;
        }

        void getAB()
        {
            cout << "a = " << a << endl;
            cout << "b = " << b << endl << endl;
        }

        ~AB() /* это деструктор. не будем заставлять
его чистить память, пусть просто покажет где он
сработал*/
        {
            cout << "Тут сработал деструктор" <<
endl;
        }
    };

    int main()
    {
    setlocale(LC_ALL, "rus");

    AB obj1(100, 100);  /*передаем конструктору
параметры*/
```

```
    obj1.setAB();   /*/присвоим новые значения
переменным*/
    obj1.getAB();   //и выведем их на экран

    AB obj2(200, 200);  /*передаем конструктору
параметры*/
    }
```

Результаты решения.

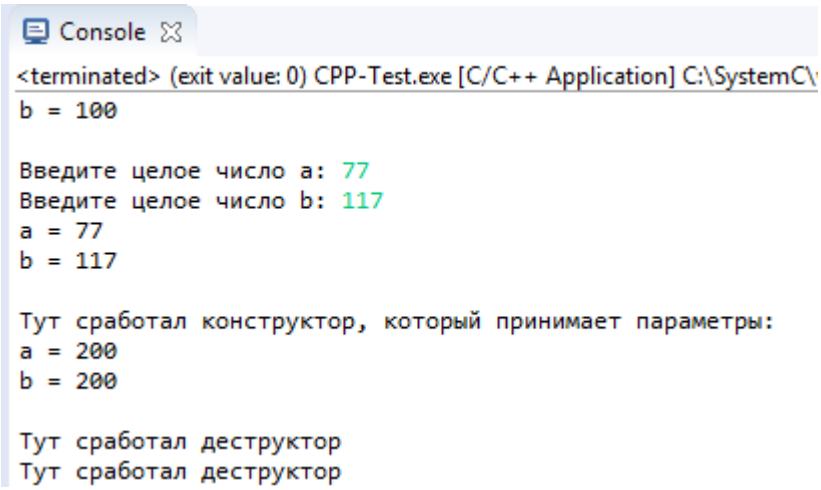


Рис. А.19. Работа конструктора и деструктора

Деструктор срабатывает в тот момент, когда завершается работа программы и уничтожаются все данные. Деструктор сработал сам. Как видно, он сработал 2 раза, так как и конструктор. В первую очередь, он удалил второй созданный объект (где a = 200, b = 200), а затем первый (где a = 100, b = 100).

А.14. Векторы в C++

Вектор в C++ — это замена стандартному динамическому массиву, память для которого выделяется вручную, с помощью оператора new.

Разработчики языка рекомендуют использовать именно `vector` вместо ручного выделения памяти для массива. Это позволяет избежать утечек памяти и облегчает работу программисту.

Пример создания вектора

```
#include <iostream>
#include <vector>

int main()
```

```
    {
        // Вектор из 10 элементов типа int
        std::vector<int> v1(10);

        // Вектор из элементов типа float
        // С неопределенным размером
        std::vector<float> v2;

        // Вектор, состоящий из 10 элементов типа int
        // По умолчанию все элементы заполняются
нулями
        std::vector<int> v3(10, 0);

        return 0;
    }
```

Методы класса vector

Для добавления нового элемента в конец вектора используется метод `push_back()`. Количество элементов определяется методом `size()`. Для доступа к элементам вектора можно использовать квадратные скобки [], также, как и для обычных массивов.

- `pop_back()` — удалить последний элемент
- `clear()` — удалить все элементы вектора
- `empty()` — проверить вектор на пустоту

A.15. Наследование классов в C++

Наследование классов — очень мощная возможность в объектно ориентированном программировании. Оно позволяет создавать производные классы (классы наследники), взяв за основу все методы и элементы базового класса (класса родителя). Таким образом экономится время на написание и отладку кода новой программы. Объекты производного класса свободно могут использовать всё, что создано и отлажено в базовом классе. При этом, мы можем в производный класс, дописать необходимый код для усовершенствования программы: добавить новые элементы, методы и т.д.. Базовый класс останется нетронутым. Ниже приведен простой код программы, который детально разберем под листингом. В этой программе примера A.17 созданы два класса: базовый — FirstClass и производный от него SecondClass.

Пример А.17

```
1     #include <iostream>
2     using namespace std;
3
4     class FirstClass    // базовый класс
5     {
6     protected: /* спецификатор доступа к
7                элементу value*/
8         int value;
9     public:
10        FirstClass()
11        {
12            value = 0;
13        }
14
15        FirstClass( int input )
16        {
17            value = input;
18        }
19
20        void show_value()
21        {
22            cout << value << endl;
23        }
24    };
25
26    class SecondClass : public FirstClass
27        //производный класс
28    {
29    public:
30        SecondClass() : FirstClass ()   /*
31 конструктор класса SecondClass вызывает
32 конструктор класса FirstClass*/
33        {}
34
35        SecondClass(int inputS) : FirstClass
36 (inputS)   /* inputS передается в конструктор с
37 параметром класса FirstClass*/
38        {}
39
40        void ValueSqr () /* возводит value в
```

```
41  квадрат. Без спецификатора доступа protected эта
42  функция не могла бы изменить значение value*/
43          {
44              value *= value;
45          }
46      };
47
48      int main()
49      {
50          setlocale(LC_ALL, "rus");
51
52          FirstClass F_object(3);     /* объект
53  базового класса*/
54          cout << "value F_object = ";
55          F_object.show_value();
56
57          SecondClass S_object(4);    /* объект
58  производного класса*/
59          cout << "value S_object = ";
60          S_object.show_value();  /* вызов метода
61  базового класса*/
62
63          S_object.ValueSqr();        /* возводим
64  value в квадрат*/
65          cout << "квадрат value S_object = ";
66          S_object.show_value();
67
68          F_object.ValueSqr();      /* базовый
69  класс не имеет доступа к методам производного
70  класса*/
71
72          cout << endl;
73          return 0;
        }
```

Рис. А.20. Наследование классов

Ранее мы работали только со спецификаторами доступа `private` и `public`. В **строке 6** использован новый для нас спецификатор доступа `protected`. Он отличается от `private` тем, что разрешает доступ к элементам базового класса из производных классов. Если бы элемент `value` находился в поле `private`, то доступ к нему был бы закрыт и мы

бы не могли изменить его значение через объект класса `SecondClass`, используя функцию `ValueSqr()`, определённую в **строках 40 — 46**.

Чтобы было наглядней, отличия спецификаторов доступа можно отобразить в таблице:

Таблица А.4

	private	protected	public
Доступ из тела класса	открыт	открыт	открыт
Доступ из производных классов	закрыт	открыт	открыт
Доступ из внешних функций и класов	закрыт	закрыт	открыт

Если вы создаёте класс, который в дальнейшем планируете использовать, как базовый, то объявляйте в нём поле `protected` вместо `private`. Иначе объекты производного класса не смогут обращаться к элементам базового.

Ниже, в **строках 10 — 24**, определены методы базового класса. Конструктор без параметров `FirstClass()`, конструктор с параметром `FirstClass( int input )` и метод `void show_value()`, который выводит значение `value` на экран.

Определение производного класса находится в **строках 26 — 46**. Синтаксис наследования такой:

class Имя_Производного_Класса : спецификатор доступа Имя_Базового_Класса { } ;

Программа:

```
    class SecondClass : public FirstClass   /*
производный класс */
    {
    public:
        SecondClass() : FirstClass ()   /*
конструктор класса SecondClass вызывает конструктор
класса FirstClass */
        {}

        SecondClass(int inputS) : FirstClass (inputS)
/* inputS передается в конструктор с параметром
класса FirstClass */
        {}
```

```
        void ValueSqr () /* возводит value в квадрат.
Без спецификатора доступа protected эта функция не
могла бы изменить значение value */
        {
            value *= value;
        }
    };
```

Двоеточие `:` не путайте с двойным двоеточием `::` (определение области действия). Используя этот оператор мы показываем, наследником какого класса является производный класс.

Важной особенностью производного класса, является то, что хоть он и может использовать все методы и элементы полей `protected` и `public` базового класса, но он не может обратиться к конструктору с параметрами. Если конструкторы в производном классе не определены, при создании объекта сработает конструктор без аргументов базового класса. А если нам надо сразу при создании объекта производного класса внести данные, то для него необходимо определить свои конструкторы. В примере показано, как можно использовать уже готовые конструкторы базового класса, чтобы не набирать код конструкторов снова — строки 26 — 46. Для этого при определении конструктора производного класса после его имени следует поставить оператор : и имя конструктора базового класса, который необходимо вызвать, при создании объекта производного класса:

```
SecondClass() : FirstClass (){}.
```

Тело конструктора оставляем пустым т.к. всю работу проделает конструктор базового класса. В случае конструктора с параметром, этот параметр мы передаем в конструктор с параметром базового класса `SecondClass(int inputS) : FirstClass (inputS){}` — **строка 35**.

В main-функции создаем объекты базового и производного классов: `FirstClass F_object(3);` и `SecondClass S_object(4);` и отображаем их значения `value` на экран. В **строке 55**, объект производного класса без проблем обращается к методу `show_value()` базового класса, будто это его собственный метод. Ниже вызываем метод, который возводит значения `value` производного класса в квадрат. И выводим это изменённое значение на экран. Если мы захотим вызвать этот метод —`F_object.ValueSqr();`– для объекта базового класса, компилятор нам этого не позволит сделать и выдаст ошибку. Это еще одна важная особенность — производный класс имеет доступ к базовому классу, а базовый класс, даже «не знает» о существовании производного и не может пользоваться его кодом.

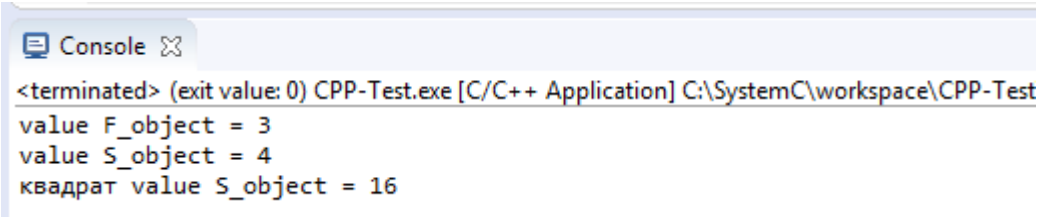


Рис. А.22. Результат работы программы:

Приведём основную информацию о наследовании классов, которую важно знать:

Наследование — это определение производного класса, который может обращаться ко всем элементам и методам базового класса за исключением тех, которые находятся в поле `private`;

Производный класс еще называют потомком или подклассом, а базовый — родитель или надкласс;

Синтаксис определения производного класса:

class Имя_Производного_Класса : спецификатор доступа Имя_Базового_Класса { /*код*/ } ;

Производный класс имеет доступ ко всем элементам и методам базового класса, а базовый класс может использовать только свои собственные элементы и методы.

В производном классе необходимо явно определять свои конструкторы, деструкторы и перегруженные операторы присваивания из-за того, что они не наследуются от базового класса. Их можно вызвать явным образом при определении конструктора, деструктора или перегрузки оператора присваивания производного класса, например таким образом (для конструктора):

```
Конструктор_Производного_Класса (/*параметры*/) :
Конструктор_Базового_Класса ( /*параметры*/) { }.
```

Важный момент при наследовании — перегруженные функции-методы класса потомка. Если в классе родителе и в его классах потомках встречаются методы с одинаковым именем, то для объектов класса потомка компилятор будет использовать методы именно класса потомка. Перегруженные методы класса потомка, могут вызывать методы класса родителя. В таком случае важно помнить, что необходимо правильно определить область действия с помощью оператора `::`. Иначе компилятор воспримет это, как вызов функцией самой себя. Например, если бы мы перегрузили в классе `SecondClass` функцию `show_value()` — это выглядело бы так:

```
void show_value()
{
if(value != 0)
```

```
    FirstClass ::
show_value();
    }
```

Эта запись указывает компилятору — если значение `value` не равно нулю — вызвать метод `show_value()` класса `FirstClass.` А он в свою очередь, отобразит это значение на экране.

Думаю для первого знакомства с наследованием классов этого достаточно. Это бесспорно классная возможность языка C++. Она помогает экономить массу времени на написание и отладку кода с нуля.

Наследование классов позволяет нам использовать уже готовый и отлаженный код и подстраивать его под новые задачи. При этом новая программа будет занимать намного меньше строк, что значительно улучшит её читабельность.

A.16. Дополнительные вопросы по C++

Перегрузка функций

Перегрузка функций в C++ используется, когда нужно сделать одно и то же действие с разными типами данных. Для примера, создадим простую функцию `max`, которая будет определять максимальное из двух целых чисел.

Перегрузка методов класса в C++.

Методы класса можно перегружать также, как и обычные функции. Особенно это удобно, когда нужно сделать несколько конструкторов, которые будут принимать разные параметры.

Например, попробуем создать основу класса `decimal,` который реализует длинную арифметику для чисел произвольной точности. В таких случаях, обычно хранят число внутри строки, а логика математических операций реализуется через написание соответствующих операторов класса.

Сделаем так, чтобы в конструктор этого класса можно было передавать и строку и число типа `double.`

Определение и перегрузка операторов класса в C++

В C++ можно определять пользовательские операторы для собственных типов данных. Оператор определяется, как обычная функция-член класса, только после определения возвращаемого типа ставится ключевое слово `operator.`

Раздельная компиляция программ на C++

Термины

Ниже даны определения терминов так, как они обычно используются.

Исходный код — программа, написанная на языке программирования, в текстовом формате. А также текстовый файл, содержащий исходный код.

Компилятор — программа, выполняющая компиляцию На данный момент среди начинающих наиболее популярными компиляторами C/C++ являются *GNU g++* (и его порты под различные ОС) и *MS Visual Studio C++* различных версий.

Компиляция — преобразование исходного кода в объектный модуль.

Объектный модуль — двоичный файл, который содержит в себе особым образом подготовленный исполняемый код, который может быть объединён с другими объектными файлами при помощи редактора связей (компоновщика) для получения готового исполняемого модуля, либо библиотеки.

Компоновщик (**редактор связей**, **линкер**, **сборщик**) — это программа, которая производит компоновку («линковку», «сборку»): принимает на вход один или несколько объектных модулей и собирает по ним исполнимый модуль.

Исполняемый модуль (**исполняемый файл**) — файл, который может быть запущен на исполнение процессором под управлением операционной системы.

Препроцессор — программа для обработки текста. Может существовать как отдельная программа, так и быть интегрированной в компилятор. В любом случае, входные и выходные данные для препроцессора имеют **текстовый формат**. Препроцессор преобразует текст в соответствии с *директивами препроцессора*. Если текст не содержит директив препроцессора, то текст остаётся без изменений..

IDE (англ. Integrated Development Environment) — интегрированная среда разработки. Программа (или комплекс программ), предназначенных для упрощения написания исходного кода, отладки, управления проектом, установки параметров компилятора, линкера, отладчика. Важно не путать IDE и компилятор. Как правило, компилятор самодостаточен. В состав IDE компилятор может не входить. С другой стороны с некоторыми IDE могут быть использованы различные компиляторы.

Объявление — описание некой сущности: сигнатура функции, определение типа, описание внешней переменной, шаблон и т.п. Объявление уведомляет компилятор о её существовании и свойствах.

Определение — реализация некой сущности: переменная, функция, метод класса и т.п. При обработке определения компилятор генерирует

информацию для объектного модуля: исполняемый код, резервирование памяти под переменную и т.д.

От исходного кода к исполняемому модулю

Создание исполняемого файла издавна производилось в три этапа: (1) обработка исходного кода препроцессором, (2) компиляция в объектный код и (3) компоновка объектных модулей, включая модули из объектных библиотек, в исполняемый файл. Это классическая схема для компилируемых языков. (Сейчас уже используются и другие схемы.)

Часто *компиляцией* программы называют весь процесс преобразования исходного кода в исполняемы модуль. Что неправильно. Обратите внимание, что в IDE этот процесс называется *построение* (*build*) проекта.

IDE обычно скрывают три отдельных этапа создания исполняемого модуля. Они проявляются только в тех случаях, когда на этапе препроцессинга или компоновки обнаруживаются ошибки.

Разделение текста программы на модули

Разделение исходного текста программы на несколько файлов становится необходимым по многим причинам:

С большим текстом просто неудобно работать.

Разделение программы на отдельные модули, которые решают конкретные подзадачи.

Разделение программы на отдельные модули, с целью повторного использования этих модулей в других программах.

Разделение интерфейса и реализации

Как только мы решаем разделить исходный текст программы на несколько файлов, возникают две проблемы:

Необходимо от простой компиляции программы перейти к раздельной. Для этого надо внести соответствующие изменения либо в последовательность действий при построении приложения вручную, либо внести изменения в командные или `make-файлы`, автоматизирующие процесс построения, либо внести изменения в проект IDE.

Необходимо решить каким образом разбить текст программы на отдельные файлы.

Во-первых, нужно определить какие части программы выделить в отдельные модули.

Во-вторых, нужно определить интерфейсы для модулей. Здесь есть вполне чёткие правила.

Интерфейс и реализация

Таким образом, модуль состоит из двух файлов: заголовочного (интерфейс) и файла реализации.

Заголовочный файл, как правило, имеет расширение *.h* или *.hpp*, а файл реализации — *.cpp* для программ на C++ и *.c*, для программ на языке C.

Заголовочный файл должен содержать все объявления, которые должны быть видны снаружи. Объявления, которые не должны быть видны снаружи, делаются в файле реализации.

Что может быть в заголовочном файле

Правило 1. Заголовочный файл может содержать только объявления. Заголовочный файл не должен содержать определения**.**

То есть, при обработке содержимого заголовочного файла компилятор не должен генерировать информацию для объектного модуля.

Единственным «исключением» из этого правила является определение метода в объявлении класса. Но по стандарту языка, если метод определён в объявлении класса, то для этого метода используется инлайновая подстановка. Поэтому, такое объявление не порождает исполняемого кода — код будет генерироваться компилятором только при вызове этого метода.

Аналогичная ситуация и с объявлением переменных-членов класса: код будет порождаться при создании экземпляра этого класса.

Правило 2. Заголовочный файл должен иметь механизм защиты от повторного включения.

Защита от повторного включения реализуется директивами препроцессора:

```
#ifndef SYMBOL
#define SYMBOL

// набор объявлений

#endif
```

Для препроцессора при первом включении заголовочного файла это выглядит так: поскольку условие "символ `SYMBOL` не определён" (`#ifndef SYMBOL`) истинно, определить символ `SYMBOL` (`#define SYMBOL`) и обработать все строки до директивы `#endif`. При повторном включении — так: поскольку условие " символ `SYMBOL` не определён" (`#ifndef SYMBOL`) ложно (символ был определён при первом включении), то пропустить всё до директивы `#endif`.

В качестве `SYMBOL` обычно применяют имя самого заголовочного файла в верхнем регистре, обрамлённое одинарными или сдвоенными

подчерками. Например, для файла *header.h* традиционно используется `#define __HEADER_H__`. Впрочем, символ может быть любым, но обязательно уникальным в рамках проекта.

В качестве альтернативного способа может применяться директива `#pragma once`. Однако преимущество первого способа в том, что он работает на любых компиляторах.

Заголовочный файл сам по себе не является единицей компиляции.

Что может быть в файле реализации

Файл реализации может содержать как определения, так и объявления. Объявления, сделанные в файле реализации, будут лексически локальны для этого файла. Т.е. будут действовать только для этой единицы компиляции.

Правило 3. В файле реализации должна быть директива включения соответствующего заголовочного файла.

Понятно, что объявления, которые видны снаружи модуля, должны быть также доступны и внутри.

Правило также гарантирует соответствие между описанием и реализацией. При несовпадении, допустим, сигнатуры функции в объявлении и определении компилятор выдаст ошибку.

Правило 4. В файле реализации не должно быть объявлений, дублирующих объявления в соответствующем заголовочном файле.

При выполнении **Правила 3**, нарушение **Правила 4** приведёт к ошибкам компиляции.

Приложение Б. Установка SystemC-2.3.1 в среде Microsoft Visual Studio 2012

Б.1. Предисловие

Материалы, видеофильмы, рекомендации, обсуждения с форумов по SystemC я собирал полгода. Многократно пытался выполнить установку версий 2.3.0 и 2.3.1 на Visual C++ 2010 и 2012. Эти попытки были безуспешными. Причины в том, что разные версии SystemC имеют свои особенности при установке на разные выпуски VC++. Кроме того, количество различных установок на вкладках VC++ при создании проекта слишком велико и часто рекомендации содержат ошибки. Нельзя забывать правильно установить «Переменные среды» и «Пути» в Windows. В конце концов, частично используя перевод последней инструкции от Accellera, которая входит в пакет загрузки SystemC-2.3.1, методом проб и ошибок смог установить SystemC-2.3.1 в среде Visaul C++ 2012 Ultimate на Windows-7 и 8. Пробы потребовались потому, что далеко не все написано ясно и однозначно.

Б.2. Системные требования и указания от Accellera

Windows-7 и-8
Уровень совместимости
Архитектуры: X86 (32-разрядная версия);
X86_64 (64-разрядная версия).
Компиляторы:
GNU C++ компилятор или совместимый.

Указания по установке для Windows от Accellera

Этот релиз был протестирован на версиях Visual C++ 2005 до 2013, работает на Windows 7.

Примечание: В данном разделе описывается установка, основанная на Microsoft Visual C++.

Для установок на основе Cygwin или MinGW, смотрите другой раздел.

Скачайте архив с сайта accellera.org. Для распаковки рекомедуется использовать 7-ZIP.

Важное замечание: SystemC предназначена для работы в различных операционных системах, с разными компиляторами. Но в Readme указаны необходимые сочетания свойств операционных сред, при которых были выполнены проверки.

Предоставленные файлы проекта подготовлены как для 32-битных 'Win32' и 64-разрядных «x64» конфигураций. Пожалуйста, обратитесь к

документации Microsoft Visual Studio для получения подробной информации о 64-битной сборке.

Мне не удалось установить SC на VC++ в конфигурации Debug Win32, так как после компиляции библиотек в папке Debug отсутствовал файл Systemc.lib.

Успешной оказалась конфигурация Debug x64.

Б.3. Создание библиотек SystemC-2.3.1

Каталог для загрузки содержит два подкаталога: 'msvc80' и 'Examples'.

'Msvc80' каталог содержит файлы проекта и рабочей области для компиляции библиотеки «systemc.lib».

В каталоге msvc80 дважды щелкните на файле "SystemC.sln". Это файл для запуска Visual C++ с файлом рабочей области. Файл рабочего пространства будет иметь соответствующие переключатели, установленные для компиляции в среде Visual C++ 2012.

Установите конфигурацию Debug x64 и Выберите Build SystemC в меню Build или нажмите F7, чтобы создать Systemc.lib.

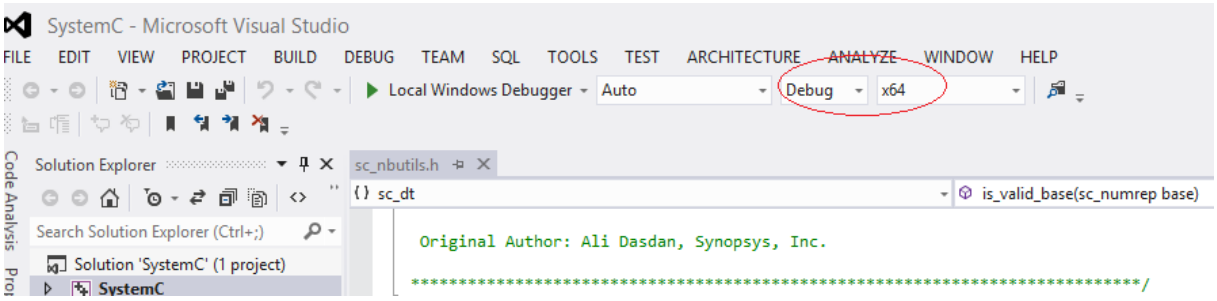


Рис. Б.1

Успешная компиляция подтверждается сообщением в консоли.

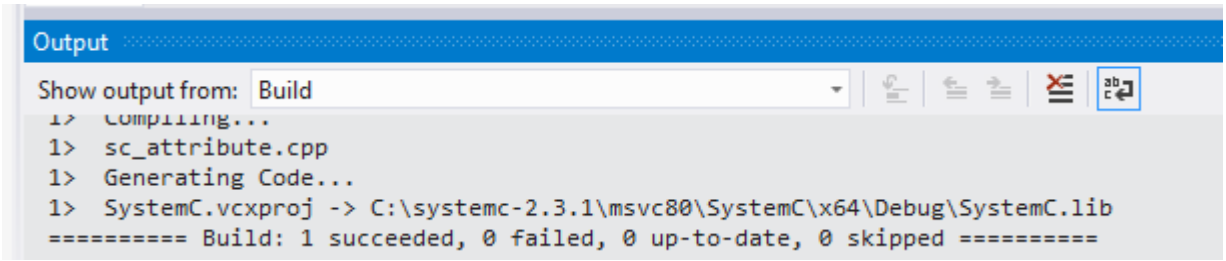


Рис. Б.2

Повторите компиляцию для Release x64.

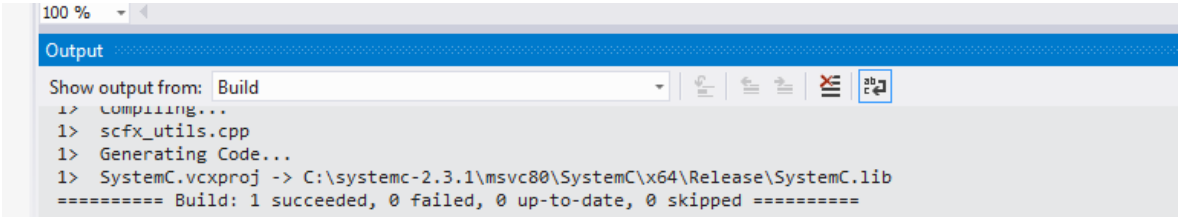


Рис. Б.3

Проверьте появление большой библиотеки в папках SystemC/…./Debug и …/Release. Обязательно надо удостовериться в наличии файла Systemc.lib со значком VC++ (рис. Б.4).

Установите свойства файла Systemc.lib (рис.Б.5).

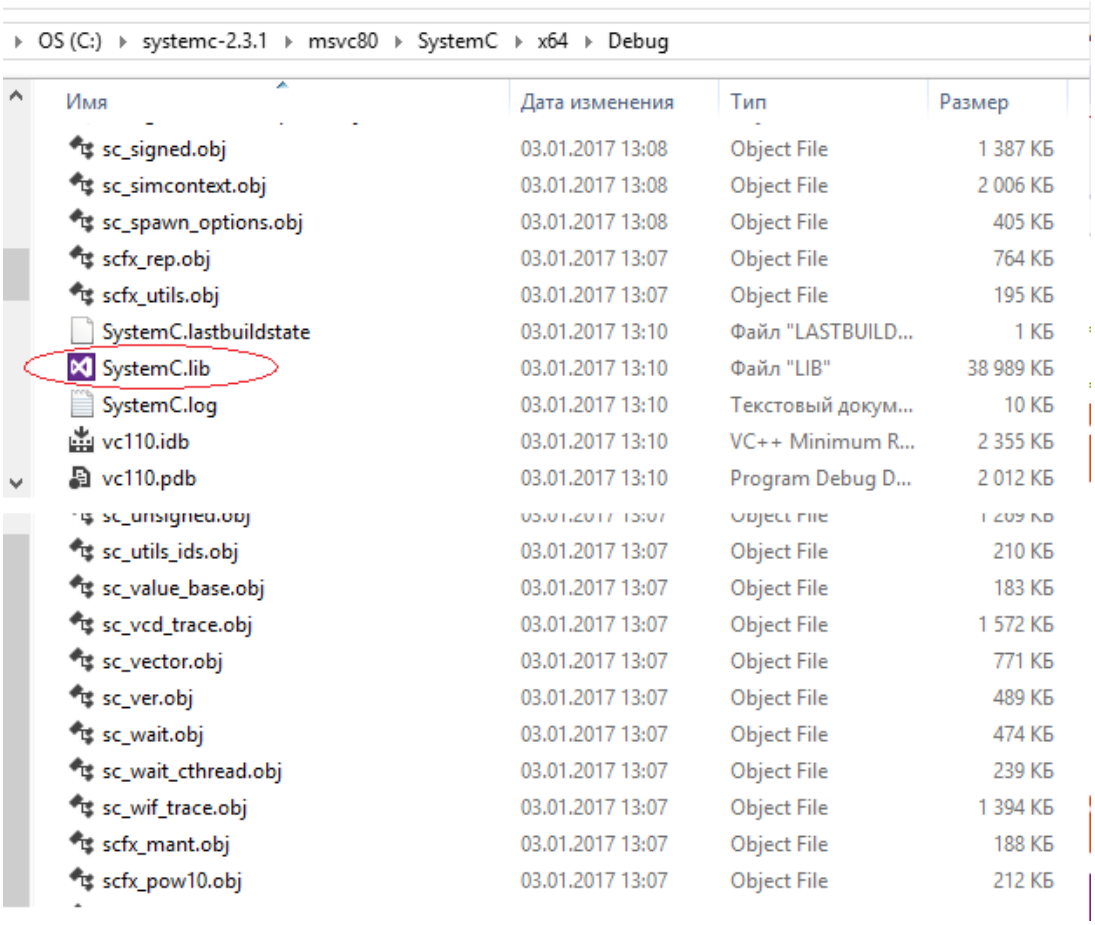


Рис. Б.4

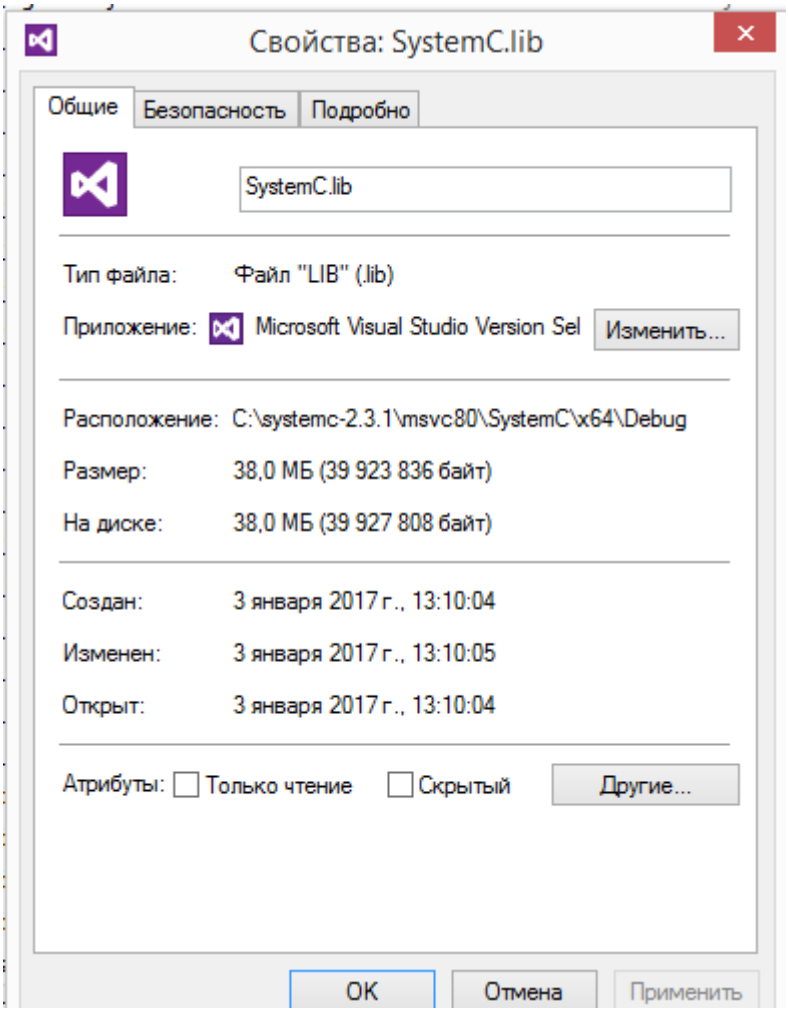


Рис. Б.5

Б.4. Системные переменные

Для указания путей к библиотекам SystemC-2.3.1 в свойствах компьютера выберите «Дополнительные параметры системы» > «Переменные среды» и установите (рис. Б.6):

Systemc.lib =C:/systemC-2.3.1/msvc80/SystemC/x64/Debug

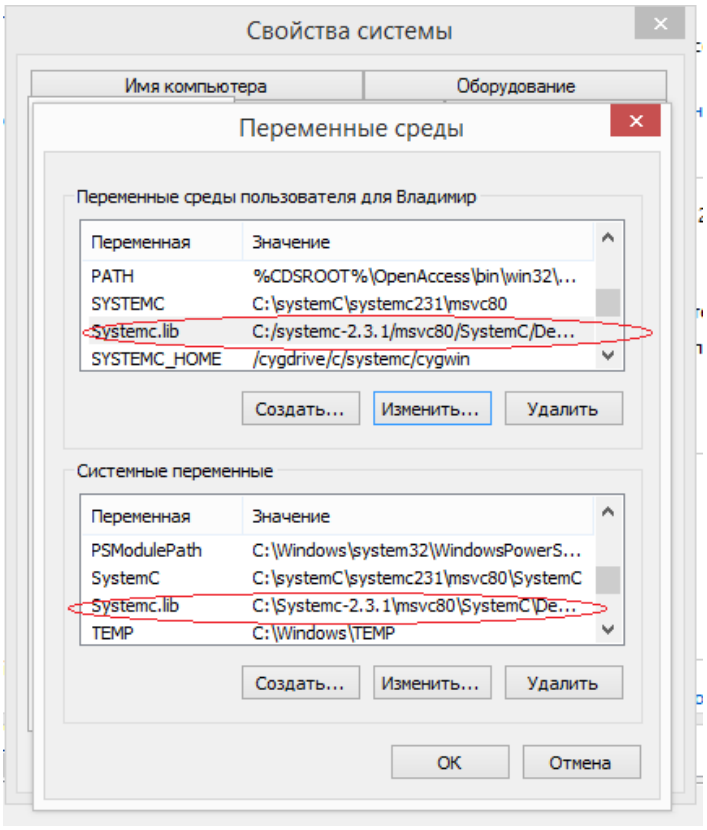


Рис. Б.6. Установка переменной среды и системной переменной

Для пользовательской переменной PATH добавляем:
C:\SystemC-2.3.1\msvc80\SystemC\x64\Debug;

Б.5. Создание SystemC Application

1. Запустите Visual Studio. Из начальной страницы выберите New Project и Win32 Console Project. Введите имя проекта и выберите подходящее место размещения, затем нажмите кнопку OK.

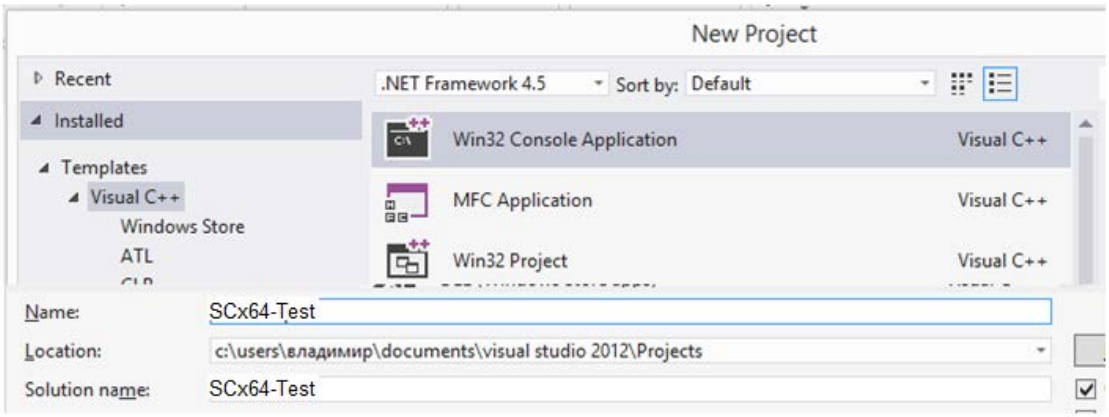


Рис. Б.7

2. Выберите страницу Application Settings of the Win32 Application Wizard .

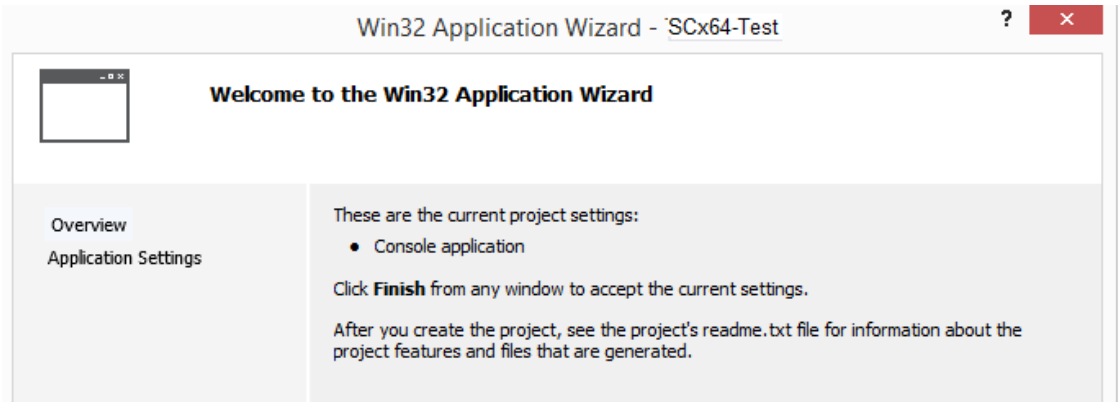


Рис. Б.8

3. Убедитесь, что поле "Empty Project" отмечено. Нажмите кнопку "Готово", чтобы завершить работу мастера.

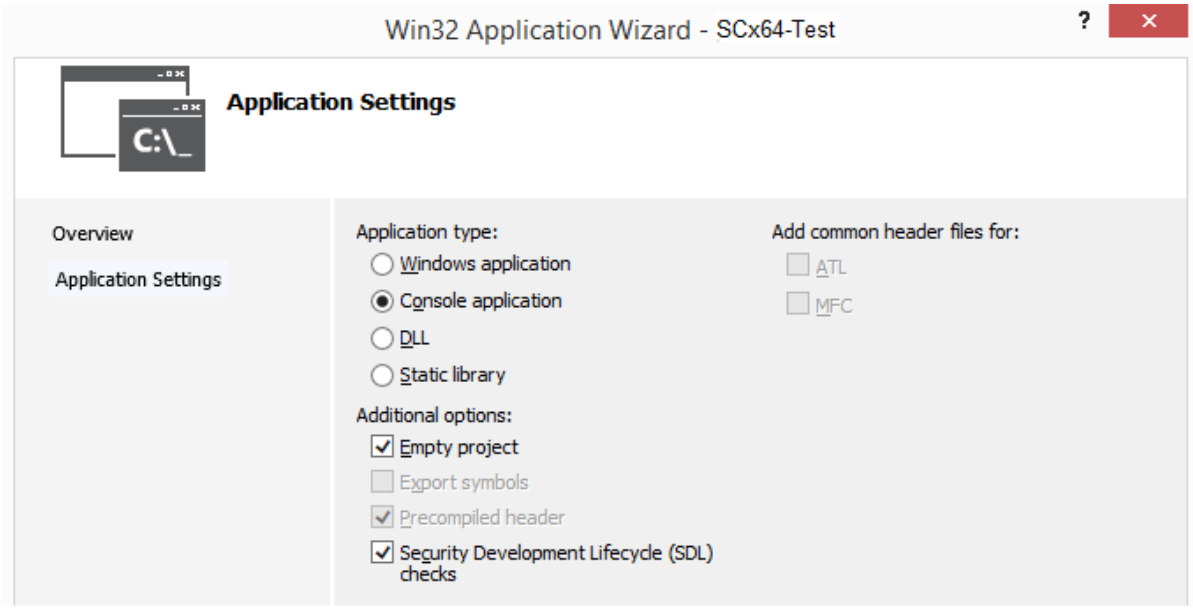


Рис. Б.9

4. Добавьте new/existing C++ файлы для проекта и отредактируйте код. В нашем примере использованы файлы из папки pipe каталога examples пакета SystemC-2.3.1.

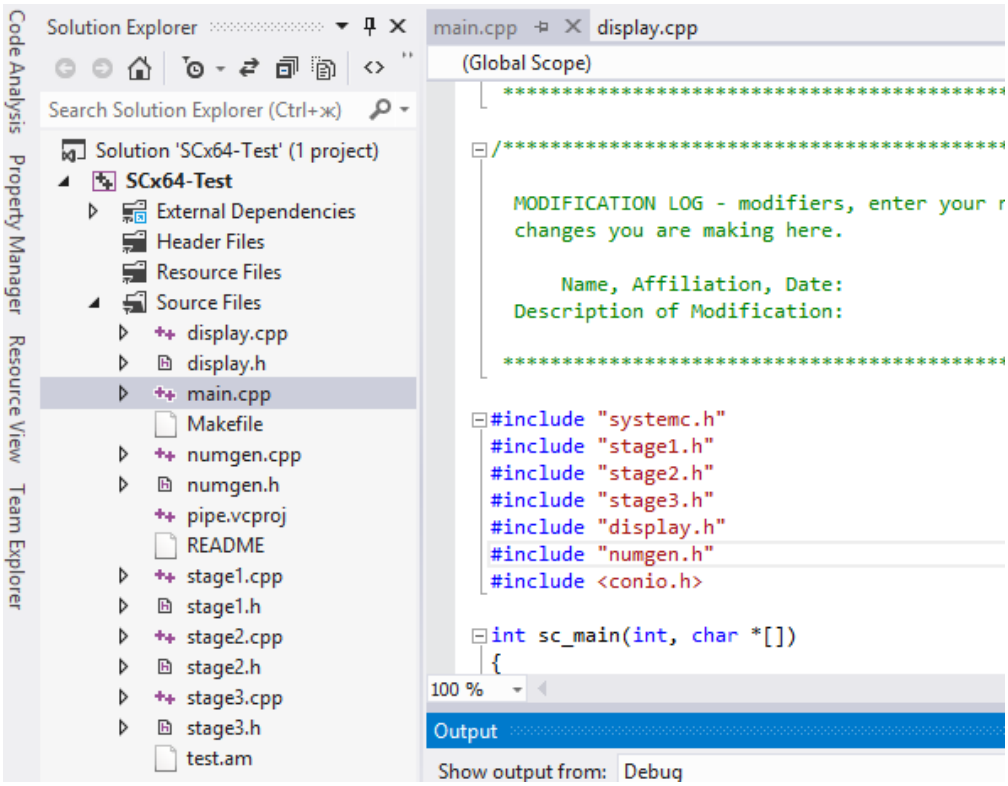


Рис. Б.10

Б.6. Установка свойств проекта

1. Отобразите Project Property Pages, выбрав пункт " Properties" из меню проекта.

2. На вкладке General установите то, что указано жирным шрифтом на рис. Б.11. Остальные установки этой и других вкладок должны сохраниться по умолчанию.

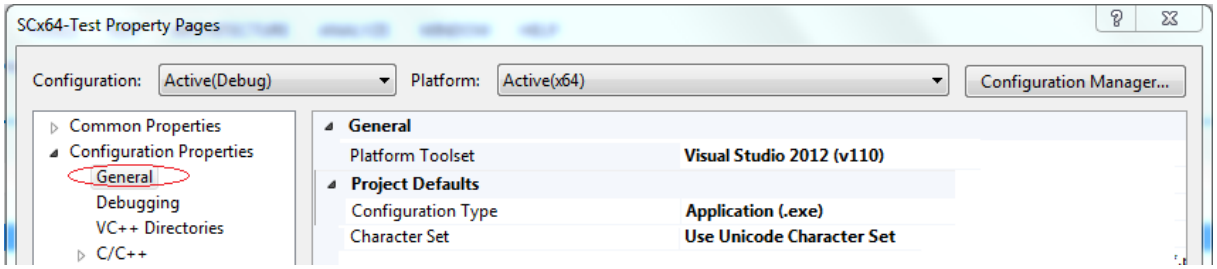


Рис. Б.11. Вкладка General

3. На вкладке VC++Directories установите (рис. Б.12):
Include Directories=C:\systemc-2.3.1\src;
Library directories=C:\systemc-2.3.1\msvc80\SystemC\x64\Debug;

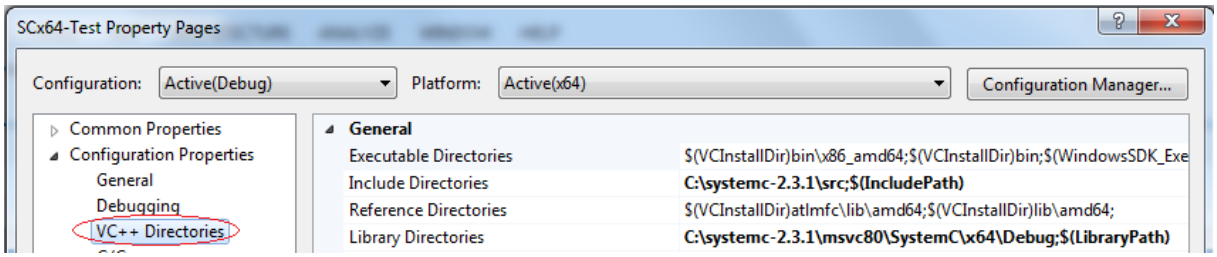


Рис. Б.12. ВкладкаVC++ Directories

4. На вкладке C/C++ >General установите (рис. Б.13):
Additional Include Directories = C:\systemc-2.3.1\src;
Warning Level= Level3(W3)
SDL checks= Yes(/sdl)

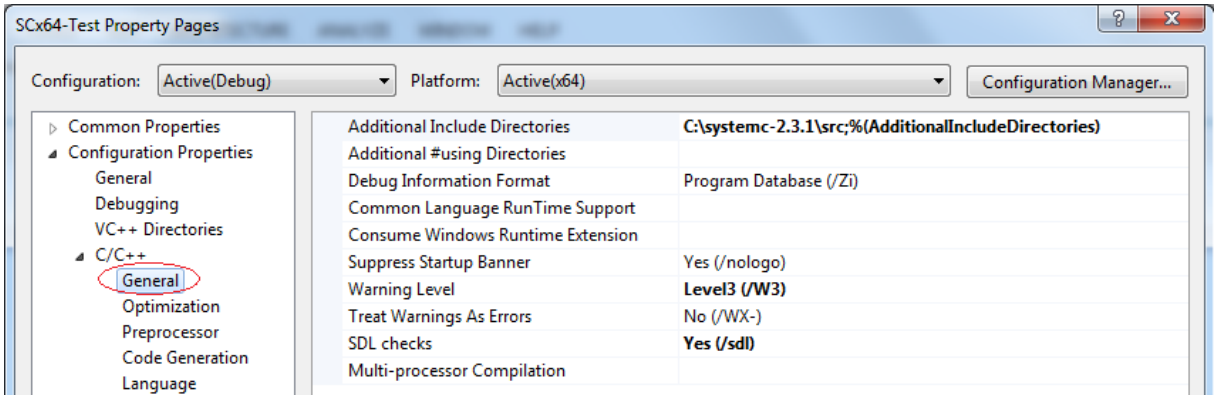


Рис. Б.13. Вкладка C/C++>General

5. На вкладке C/C++>Optimization установите (рис. Б.14):
Optimization=Disable(/Od):

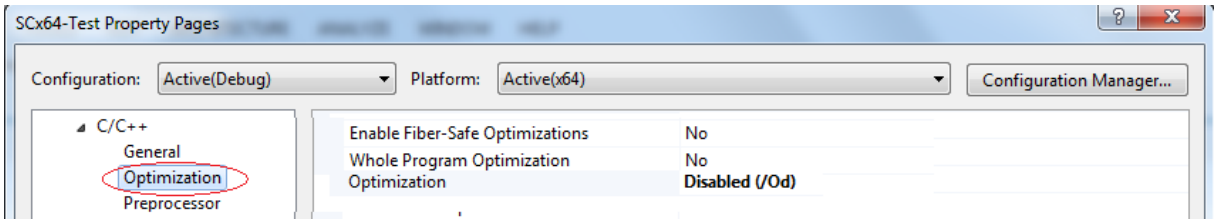


Рис. Б.14. Вкладка C/C++>Optimization

6. На вкладке C/C++>Preprocessor установите (рис. Б.15) :
Preprocessor Definitions=WIN32;_DEBUG;_CONSOLE;

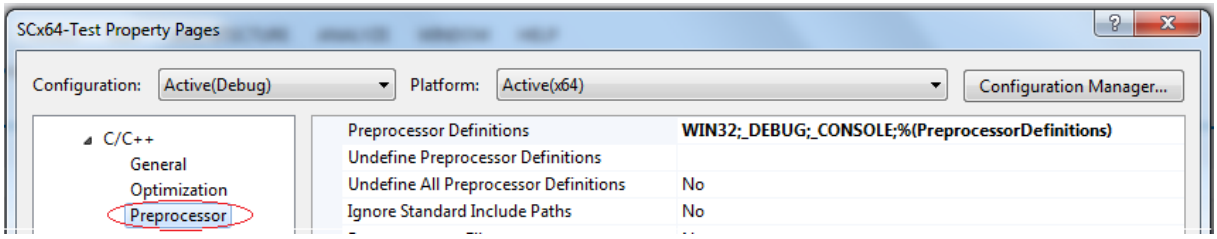


Рис. Б.15. Вкладка C/C++>Preprocessor

7. На вкладке C/C++>Code Generation установите (рис. Б.16):
Runtimed Library=Multi-threaded Debug (/MTd);

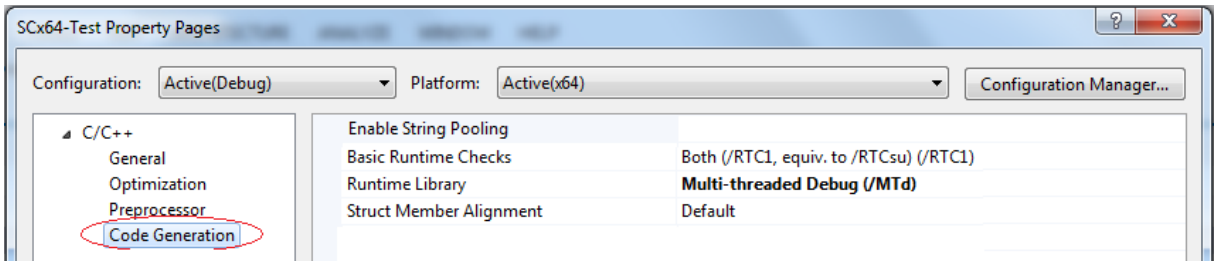


Рис. Б.16. Вкладка C/C++>Code Generation

8. На вкладке C/C++>Language установите (рис. Б.17):
Enable Run-Time TypeInformation = Yes (GR)

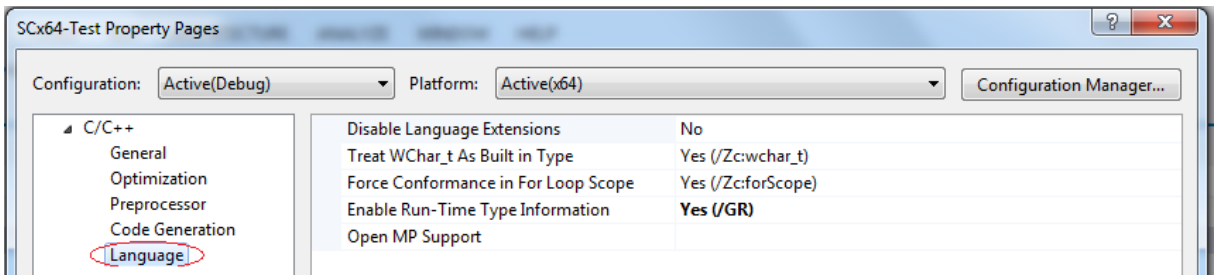


Рис. Б.17. Вкладка C/C++>Language

9. На вкладке C/C++>Command Line введите Additional Option =/vmg (рис. Б.18).

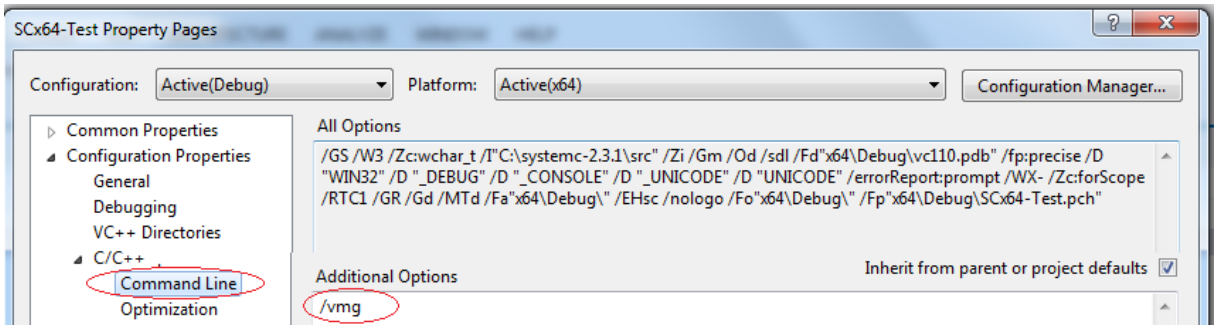


Рис. Б.18. Вкладка C/C++>Command Line

10. На вкладке Linker>General установите (рис. Б.19):
Enable Incremental Linking = Yes (INCREMENTAL)

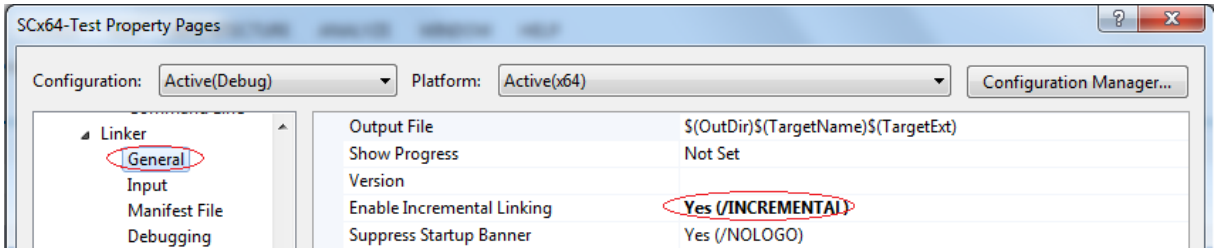


Рис. Б.19. Вкладка Linker>General

11. На вкладке Linker>Input установите (рис. Б.20):
Additional Dependencies = systemc.lib;

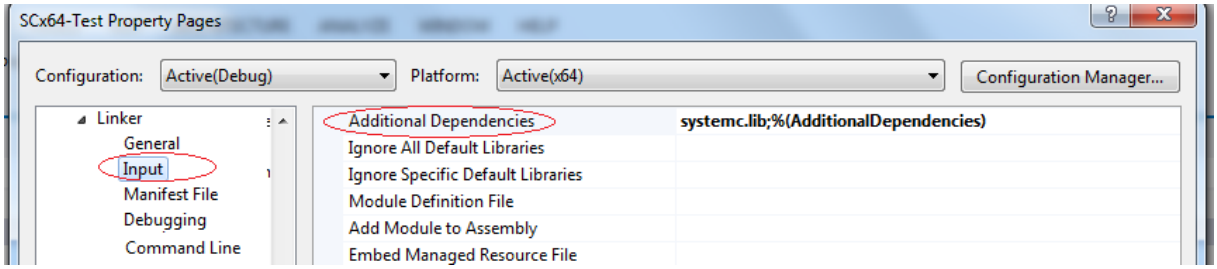


Рис. Б.20. Вкладка Linker>Input

12. На вкладке Linker>Debuggind установите (рис. Б.21):
Generate Debug Info = Yes (/DEBUG)

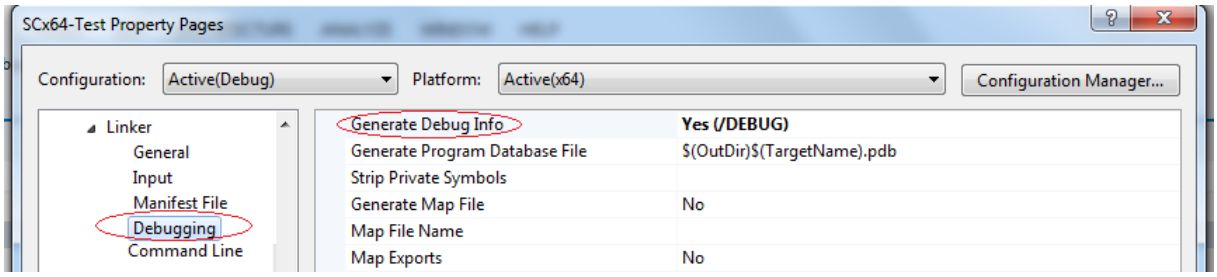


Рис. Б.21. Вкладка Linker>Debuggind

13. На вкладке Linker>System установите (рис. Б.22):
SubSystem =Console (/SUBSYSTEM:CONSOLE)

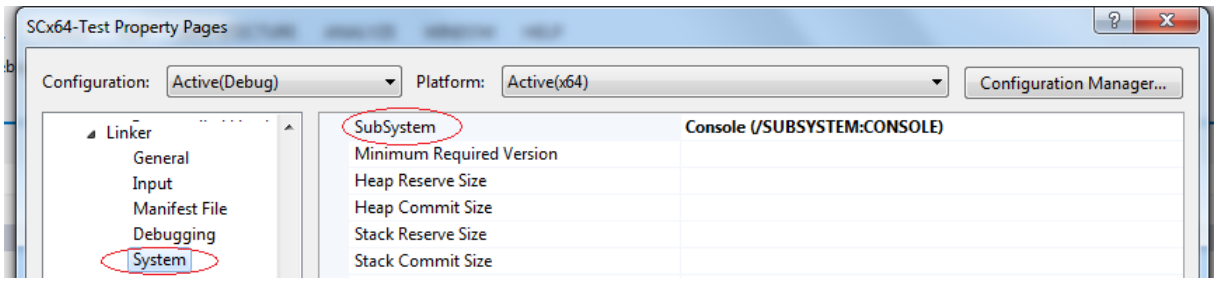


Рис. Б.22. Вкладка Linker>System

Б.7. Компиляция и проверка проекта

1. После первой компиляции проекта выявлена ошибка в библиотечном файле `sc_nbutils.h`. Команда `std::sprintf_s` не соответствовала каталогу `<stdio>`. Устранил ошибку, убрав `std::` и сохранив измененный библиотечный файл.

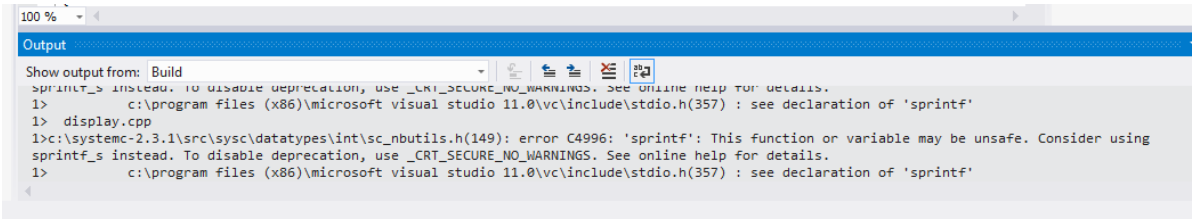


Рис. Б.23

2. Потребовалость изменить платформу в Configuration Manager, установить x64.

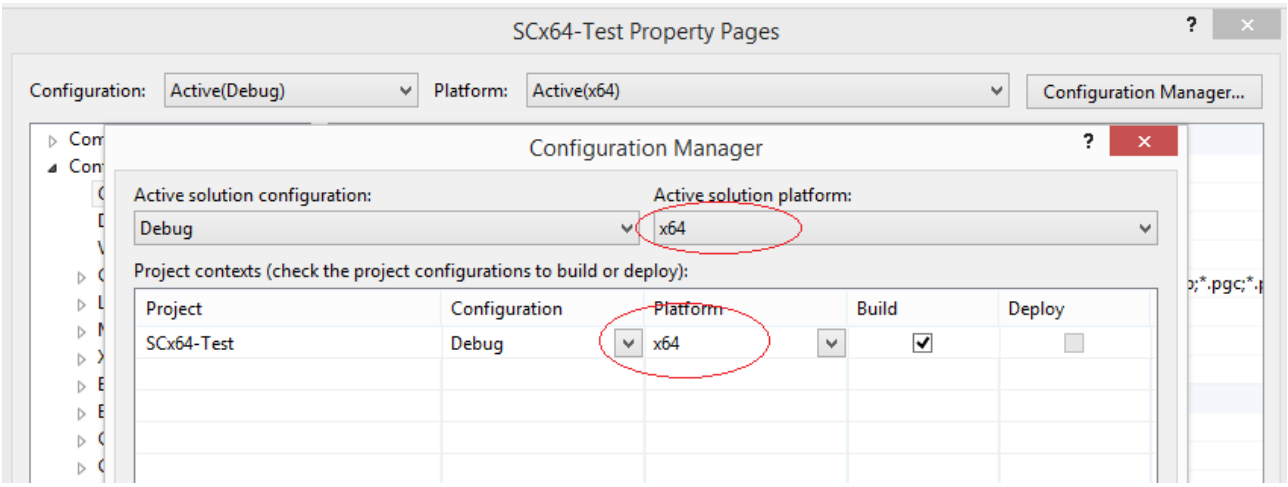


Рис. Б.24

3. Для отображении результатов в консоли в программу добавим #include <conio.h> и _getch().

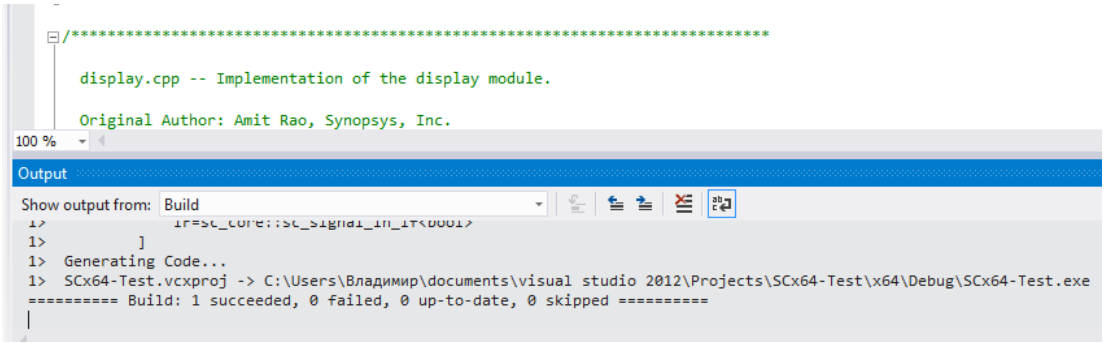


Рис. Б.25

Компиляция прошла успешно.

5. Выполним Start Debbuging и получим решение.

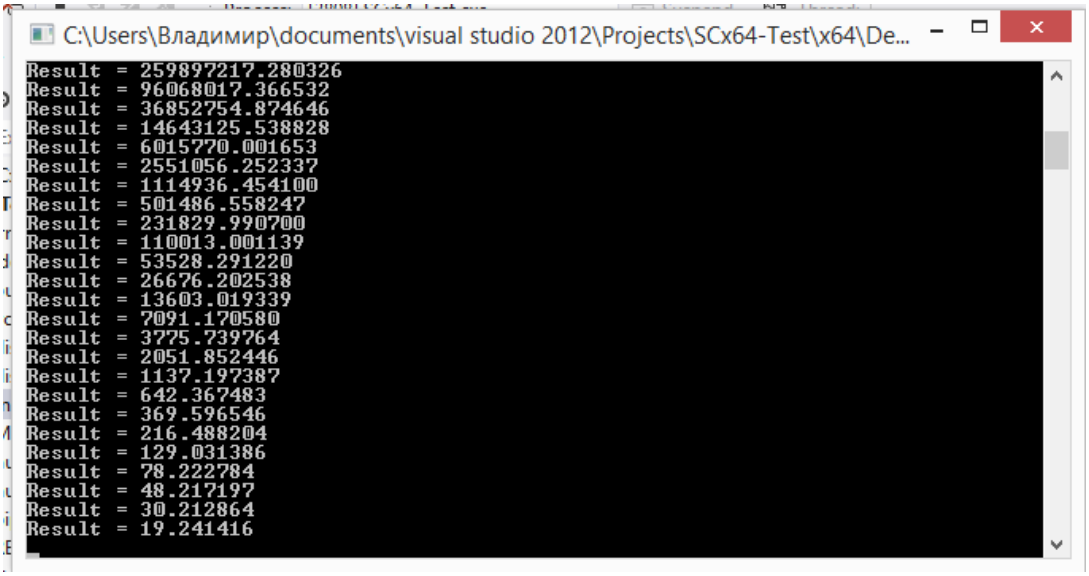


Рис. Б.26. Результаты решения

Библиография

1. Немудров В., Мартин Г. Системы-на-кристалле. Проектирование и развитие. - М.:Техносфера, 2004. –216 с

2. Губарев В.А. Методология проектирования конечных изделий, включающих вычислительные машины и комплексы, на основе СБИС класса "Система на кристалле" с использованием высокоуровневых системных моделей. Докторская диссертация.- М. :МИРЭА, 2012. - 163 с.

3. J. Bhasker. A SystemC Primer. –USA: Star Galaxy Publishing, 2002- p.283.

4. SystemC. Utilizing SystemC for Design and Verification. /Co-Authored by Alan Ma and Allan Zacharda./- USA : Mentor Grapfics. -2005, p.33

5. Thorsten Grotker, Stan Liao, Grant Martin, Stuart Swan.. System Design with SystemC. - New York, Boston, Dordrecht, London, Miscow. : Kluwer Academic Publishers, 2002.- p.236

6. David C. Black, Jack Donovan, Bill Bunton, Anna Keist. SystemC: From the Ground Up. Second Edition. -New York , Dordrecht Heidelberg London.: Springer. - 2010, p.291.

7. Fuctional specification for SystemC 2.0.- April 5, 2002. [Электронный ресурс]. – Режим достура: http://www.systemc.org

8. Amal Banerjee. SystemC and SystemC-AMS in Practice. SystemC 2.3, 2.2 and SystemC-AMS 1.0. / Amal Banerjee, Balmiki Sur. - New York Dordrecht London. : Springer Cham Heidelberg, 2014. - p.462.

9. SystemC Tutorial- [Электронный ресурс] – Режим доступа: http://www.asic-world.com/systemc/index.html

10. Introduction to Accellera TLM 2.0. // Araida Thimmapuram 10th Sept 2015. Design and verifucation conference and exhibition. India.-2015

11. Devalapalli, Siddhartha, Development of SystemC Modules from HDL for System-on-Chip Applications. : Master's Thesis, University of Tennessee, 2004. - Режим доступа: http://trace.tennessee.edu/utk_gradthes/2119

12. OSCI TLM Working Group. OSCI standard for SystemC TLM. [Электронный ресурс] – Режим доступа: www.systemc.org

13. Wolfgang Ecker,Wolfgang Müller, Rainer Dömer. Hardware-dependent Software Principles and Practice. : Springer Science + Business Media B.V. - 2009. [Электронный ресурс]. – Режим доступа: http://www.springer.com

14. Describing Synthesizabl RTL in SystemC. -Printed in the U.S.A.. : Synopsys, Inc., 2001. - p. 116.

15. Yuan Wen and Hani Mohamed Khalil. SystemC-based electronic system level design methodology for SoC design-space exploration. // In: Advances In Microelectronics. Penerbit UTM, Skudai, Johor Bahru, 2008. - pp. 9-34.

16. Frank Ghenassia. Transaction Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems. STMicroelectronics, France. - Printed in the Netherland. : Springer, 2005. - p.7.

17. OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL. Software version: TLM 2.0.1 Document version: JA32.: Open SystemC Initiative (OSCI). 2009. – p.194.

18. John Aynsley. Getting Started with TLM-2.0: Doulos.Ltd. -2017. [Электронный ресурс]. -Режим доступа:
https://www.doulos.com/knowhow/systemc/tlm2/tutorial__1/

19. Ali Habibi. A Framework for System Level Verification:The SystemC Case. // A Thesis in The Department of Electrical and Computer Engineering. Presented in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy at Concordia University. - Montr´eal, Qu´ebec, Canada. 2005.

20. Usage of System Level Modeling with SystemC. - Danish Technological Institute. 2008, p.25

21. GTKWave 3.3 Wave Analyzer User's Guide. Updated December 5, 2016. [Электронный ресурс]. –Режим доступа: http://www.gnu.org

22. Adam Rose, Stuart Swan, John Pierce, Jean-Michel Fernandez . Transaction Level Modeling in SystemC. : Cadence Design Systems, Inc. 2006. - p.16.

23. Guide for getting started with SystemC development. : Danish Technological Institute, 2007. - p.27

24. SystemC Compiler RTL User and Modeling Guide. Version 2001.08. - Printed in the U.S.A. : Synopsys, 2001. –p. 222.

25. Karsten Einwich, Thilo Vörtler, Thomas Arndt. New technological opportunities coming along with SystemC/SystemC AMS for AMS IP Handling and Simulation. – Fraunhofer. : IIS/EAS. – 2015. [Электронный ресурс]. – Режим доступа:
http://www.ti.uni-tuebingen.de/uploads/media/ESCUGWS27_fhg.pdf

26. Кустарев П.В., Ключев А.О.. Маршруты проектирования «Систем на кристалле». // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики, 2010, № 1(65).-с.93-100.

27. George Kalo. Functional Verification with SystemC.- Magnus Ljung, Integrated Systems Scandinavia AB. Christer Albinsson, KTH Syd. 2006, p.29

28. Оленев В. Л.. Моделирование на языке SystemC в процессе разработки протоколов передачи данных. // Известия высших учебных заведений. Поволжский регион. № 4 (12), 2009. –с.60-70.

29. Hoang M. Le Daniel Große Rolf Drechsler. Automatic Fault Localization for SystemC TLM Designs. //Source: IEEE Xplore Conference: Microprocessor Test and Verification (MTV), 2010 11th International Workshop on.

30. Preeti Ranjan Panda. SystemC A modeling platform supporting multiple design abstractions. - Montr´eal, Qu´ebec, Canada. : Synopsys Inc. ISSS'01, October 13, 2001.

31. Алехин В.А. Микроконтроллеры PIC. Основы программирования и моделирования в интерактивных средах MPLAB IDE, microC, TINA, Proteus. Практикум. –М.: Горячая линия – Телеком, 2016. -248 с.

32. Корчагин Ю.Э. Программирование на языках С и С++. Лабораторный практикум: учебное пособие. - Воронеж: ГОУВПО «Воронежский государственный технический университет», 2009. 251 с.

33. Уроки программирования на С++. [Электронный ресурс].- Режим доступа: -https://code-live.ru/tag/cpp-manual/ .